```
┌-----+-----┐
| B U G S |
└-----+-----┘
```

The Brown University Graphics System[1]

META 4 B / SIMALE / VECTOR GENERAL

Concepts and Facilities

Paul Constantine Anagnostopoulos

Harold Henry Webber, Jr.

John Zahorjan

The Brown University Graphics Project
Division of Applied Mathematics
Box F
Brown University
Providence, Rhode Island 02912

Updated: January 12,1976

Printed: December 6, 1976

[1] This Research is being supported by the National Science Foundation Grant GJ-28401X, the Office of Naval Research, Contract N00014-67-A-0191-0023, and the Brown University Division of Applied Mathematics; Principal Investigator Andries van Dam.

I sometimes feel, in reviewing the evidence on the design of computing systems, that the necessary conclusion is that de-kludging just is not possible. It is difficult to conceive of a mechanism which can satisfy the conditi,ns necessary for it. Nevertheless, in spite of such evidence against it, de-kludging does sometimes occur.

--adapted from Karl S. Lashley, 1950

TABLE OF CONTENTS

1 INTRODUCTION

1.1 OVERVIEW

The Brown University Graphics System:

> "The stated objectives of the project's activities are an investigation into the area of medium-cost, microprogrammable, intelligent graphics terminals and the "division of labor" trade-offs between a mainframe processor and the intelligent satellite. In addition to these goals, we are also interested in examining the impact which microprogramming has on the design of other aspects of a graphics terminal, for example, system configuration and the local operating system design."
>
> --George M. Stabler
> The BUGS Overview

The META 4B / SIMALE / Vector General comprise the core of the graphics facility of the Brown University Graphics System (BUGS). The purpose of this document is to present the concepts and facilities of this core in such a way as to make it easy to learn and pleasing to use by people at all levels of design and implementation.

In order to accomplish this goal, BUGS has herein been formalized and conceptualized beyond the level done in other documents pertaining to the system (a list of such documents is presented in references). It is hoped that by so doing, the various facilities can be made more understandable, more useful, and hence more enjoyable. If this should turn out not to be the case, however, any comments, suggestions, etc., would be greatly appreciated and carefully considered.

1.2 SYSTEM COMPONENT CONCEPTS

How are the hardware components from which BUGS is constructed interconnected and how do they interact? All components can be divided into one of two classes, known as units and stores.

1.2.1 UNITS

Units are those components of the system capable of performing manipulation of and computation upon data. In this way, a unit is an active component whose purpose is to allow a person to perform the operations necessary to solve his problem. All units can, in turn, be divided into two sub-classes, known as processors and input/output units.

1.2.1.1 PROCESSORS

Processors are units whose behavior is capable of being controlled and changed at will, that is, they are programmable. Thus the word processor is used in the conventional sense to denote a "computer" or a "CPU". A processor consists of hardware which is capable of executing a set of primitive commands, known as host instructions, which can be used directly by programmers to implement their applications.

1.2.1.2 INPUT/OUTPUT UNITS

An input/output (I/O) unit is a hardware device which is not programmable, hence having a fixed function (e.g., a card reader or a console terminal). Although I/O units are capable of performing some manipulation of data, their principal function is to transmit and present data to other system components and to human users.

1.2.2 STORES

A store is a passive component of the system whose sole capability is that of retaining or remembering data. It is the conventional memory space of a computer, although various types of stores with various operating speeds may be present on the system. Examples of stores on BUGS are main core storage and secondary disk storage.

Each of the above components will be described in greater detail below. Simply keep in mind the simple picture of stores containing programmer-defined data structures; these data structures are operated upon in a fixed manner by I/O units, and in a variable manner by the processors.

1.3 COMPONENT DESCRIPTIONS

The diagram on the following page pictures the various components of BUGS and their interconnections. Data is, in most cases, transferred among components in groups of sixteen bits, called halfwords. Furthermore, most stores are halfword-oriented, also containing data in 16-bit groups, each halfword accessable via an address specified by a number from zero to n. (The exception is the SIMALE, as described later.) The following paragraphs describe the components in greater detail.

1.3.1 PROCESSORS

There are three processors in BUGS: the META 4A, META 4B, and the Super-Integral Multi-purpose Arithmetic/Logic Expediter (SIMALE). Each processor has three stores for its own internal use, and may be connected to a variety of I/O units.

As was previously mentioned, each processor is capable of executing a set of primitive host instructions. Programs composed of these intructions reside in a program memory known as control store. It is from control store that the processor fetches, decodes, and executes host instructions.

Programmers using host instructions require work space in which to keep operands, temporary results, etc., and with which to communicate to other units. This work space comes as a set of halfwords known as a register file. Each processor has its own register file.

Finally, each processor is equipped with a relatively large local store for retaining larger amounts of information such as tables, matrices, or data lists. Although smaller than main store, local store is at least an order of magnitude faster, and hence should be used for retaining often-used information.

1.3.1.1 META 4A

The META 4A processor is composed of a Digitial Scientific Corp. META 4 computer. The META 4A is equipped with control store consisting of 4K[2] halfwords of read-only memory (ROM), making modification of host programs difficult. The register file consists of 32 halfword registers, many of which serve special purposes. Local store has not yet been implemented and is unavailable to the programmer.

The I/O units connected to the META 4A include a disk controller, a console terminal, a control panel, and a general-purpose binary switch. In addition, the META 4A is connected via a multiplexor channel to a S/360-67.

1.3.1.2 META 4B

The META 4B processor with its stores is identical in nature to the META 4A, although it is equipped with 1K halfwords of local store.

The only I/O unit connected to the META 4B is the Vector General (VG) high-speed CRT tube used for graphical display.

1.3.1.3 THE SIMALE

The SIMALE is a high-speed unit which is actually composed of four independent sub-processors. Originally intended to be used solely for the matrix operations necessary for graphical transformations, it has evolved into a completely general-purpose processor. It is equipped with 256 halfwords of fully readable-writeable control store which contains the host program. In addition, each sub-processor has a register file with three 18-bit registers, and a local store with sixteen 18-bit locations. There are no I/O units connected to the SIMALE.

[2] "K" stands for "times 1,024".

It should be clear from the diagram that data paths exist between the SIMALE and the META 4B, and between the META 4B and the META 4A.

1.3.1.4 MAIN STORE

Main store is the central data memory for the system. It is accessable from both the META 4A and META 4B processors, so that it can contain data structures, programs, etc. It can also be accessed directly by the disk controller so that data transfers can be made directly to and from main store without processor intervention. A halfword can be transferred to or from main store in 900 nanoseconds.

We are currently equipped with 32K halfwords of main store.

1.3.1.5 DISK STORE

Disk store is implemented on large circular packs, each of which contains 512,000 halfwords. These packs are removable and replaceable, hence allowing virtually any amount of backup storage. However, data is accessed via the disk controller I/O unit, and the access time is extremely slow, averaging approximately 250 milliseconds.

1.4 SYSTEM STRUCTURE

It has been said that, given the components described above, a user/programmer could implement his applications using the host instructions provided by the processors in conjunction with stores and I/O units. This would be extremely crude, however, given the rather primitive nature of these instructions, the fixed ROMs in the META 4A and META 4B, and the lack of programming facilities in general.

To alleviate this problem, the designers of BUGS have provided the user with various facilities to aid him in his work. These facilities are embodied in an overall system structure; this structure encompasses various conceptual levels into which the facilities fall. Each level has the use of those facilities supported by the lower levels, and in turn offers various additional facilities to the levels above it. The system structure is pictured below and described in the following paragraphs.

1.4.1 USER

At the pinacle of the structure is the user himself, provided with all the facilities supported by the levels below him. All in all, the level at which he can design and implement is much closer to the problem description than if he were to work on the bare hardware. He comes in contact with the following three levels:

1.4.2 PROGRAMMING LANGUAGES

The programming language is the user's vehicle for expressing the computations he wishes to perform. Ideally this programming language would be full English, however current technology allows only simple artificial languages, ranging across a spectrum starting with such high-level languages as PL/I and continuing down to a language which is directly executed by the hardware (i.e., the host language). On BUGS we will have a high-level language called ALGOL W, and currently have a low-level PL/360-like language called PL/BUGS, and assembly language, to be described later.

1.4.3 MONITOR

The monitor, also called the operating system, is a comprehensive package of programs provided to the user for performing standard and often-used functions. These functions include I/O unit control, management of storage space, program control, etc.

1.4.4 EXTENDED MONITOR

The extended monitor comprises an extension to the monitor which is specifically oriented toward the application with which the user is involved. Such extensions might include graphical support packages, scientific subroutines, or communications programs. The extended monitor provides those useful facilities which do not belong in the standard monitor because they are not generally used by all applications.

1.4.5 Q-INTERPRETER

As we have said, the user expresses his algorithms in a programming language, as opposed to directly in host instructions. Clearly then, we must have a program, called a compiler, which takes the user's programs and translates them into a sequence of host instructions. This is a very difficult task, however, due to the primitiveness of the host; current compiling techniques dictate that such compilation would be extremely inefficient and time-consuming. We might then be forced to design programming languages that were lower-level, closer to the host -- in the worst case we could require the user to specify each host instruction individually, something which we stated would be unreasonable.

In order to make high-level languages possible, and in order to provide the user with a reasonably useful language in the absence of a high-level one, we have provided a q-interpreter (commonly called an emulator) to act as an interface between the host machine and the programming language. The q-interpreter, written in the host instructions and residing in control store, provides facilities which are much more useful than the host itself. Such facilities may include the interpretation of higher-level intructions or data structures, control of I/O units, control of communication with other processors, etc. In other words, the q-interpreter provides the well-known assembly language instruction set.

Ideally, many different q-interpreters could exist, one for each application, one for each high-level languages to compile into, etc. Each of these q-interpreters could be designed so as to be well-suited for its intended purpose. However, the existence of ROM for control store rules this out, except in the case of the SIMALE (where it is fully exploited, as described later). Therefore, a q-interpreter had to be designed which was useful for all applications and all programming languages, obviously a hopeless task.

1.4.6 EXTENDED Q-INTERPRETER

In order to help alleviate the unadaptability of the ROMs, an extra level has been added between the q-interpreter and the programmming language. This level, implemented using the facilities provided by the q-interpreter, extends the q-interpreter by supporting additional facilities in a manner transparent to the programming language. Since the extended q-interpreter is programmable just like higher levels of the structure, features can be added with ease, debugged, experimented with, etc. However, the programming language uses them just like any other feature, and hence need not know that they are not really part of the q-interpreter. Eventually, when a feature is completely implemented, it can be moved down into the q-interpreter, causing an increase in speed with no reprogramming necessary.

Due to their implementation, each level falls into one of three categories, depending upon its "solidity". The most solid is the hardware, changeable only via engineering modifications. Next, the q-interpreter, although changeable, is "firm", both because it may reside in ROM and because it is somewhat difficult to program. Finally, the remaining levels, the majority thereof, are "soft" -- easily programmable and adaptable.

1.5 ONWARD

The remainder of this document is devoted to describing the META 4B / SIMALE / Vector General units of BUGS. The next few chapters describe the facilities provided by the META 4B q-interpreter.

2 DATA FACILITIES

2.1 INTRODUCTION

The META 4B, with its q-interpreter, becomes a general-purpose processor. It provides many storage and data types to the programmer with which he can design and implement any type of data structures and data bases necessary for his application. In order to operate upon this data, a comprehensive set of instructions is provided, which can be used directly in assembly language or via a high-level language and its compiler.

2.2 PROGRAMMER STORES

There are four types of stores provided to the META 4B programmer: register file, local store, main store, and the VG register file. These stores are described briefly in the following paragraphs; more detailed information is given in the course of this document.

2.2.1 REGISTER FILE

The programmer, rather than using the processor's register file, is provided with a more extensive one of his own. This register file consists of 64 halfword registers, divided into four groups of sixteen each.

The first group, numbered 0 - 15, is called the general-purpose registers (GPR). These registers can be used to contain numeric data for purposes of arithmetic or comparison, address data, program flags, etc. They are the major store for performing data operations, and can be referenced by all instructions.

The second group, numbered 16 - 31, is the control registers. These registers are used by the q-interpreter to control the execution of a user's program. Access to these registers might be useful to the programmer, and hence they are included in his register file.

The third group, numbered 32 - 47, is the ET CETERA instruction registers. Refer to Chapter 16. for an explanation of this group.

The final group, numbered 48 - 63, is the SIMALE external registers, used as a communication area between the programmer and the SIMALE processor.

2.2.2 LOCAL STORE

The META 4B is equipped with a 256-halfword local store (soon to be expanded to 1K). This local store contains often-used data, both for the user and the q-interpreter. The contents of the first 96 locations are pre-defined, as follows:

The first sixteen halfwords, numbered 0 - 15, correspond one-for-one with general-purpose registers 0 - 15.

Locations 16 - 31 are used by the q-interpreter for a data queue between the SIMALE and the VG. This queue is necessary in order to maintain a high rate of display on the VG.

Locations 32 - 47 correspond one-for-one with the ET CETERA instruction registers 32 - 47.

Locations 48 - 63 correspond one-for-one with the SIMALE external registers 48 - 63.

Locations 64 - 95 contain the information necessary to maintain the swapping of SIMALE virtual control store pages to and from real control store. These 32 halfwords are called the SIMALE virtual control store page table.

All local store locations from 96 on up can be used by the programmer for any purposes he desires.

2.2.3 MAIN STORE

The META 4B, along with the META 4A, has access to main store which is equipped with 32K halfwords. These halfwords comprise the major store for both the user's data structures and his programs.

There is a major difference between the hardware operation of main store and the way the programmer uses it via the q-interpreter. Instead of addressing a set of halfwords, the programmer addresses sequential groups of eight bits, called bytes. Each byte is assigned an address starting with zero and continuing up to ~~48~~K (currently). In effect,
64

then, the low-order bit of an address specifies one of two bytes within the halfword addressed by the remaining bits.

In spite of this added flexibility, the q-interpreter requires that certain 16-bit data be located at an even byte address, i.e., not cross a halfword boundary. This requirement is called halfword alignment.

2.2.4 VECTOR GENERAL REGISTER FILE

The VG display unit is equipped with a register file containing 85 registers of varying sizes (none greater than 16 bits). These register are used to control the display, handle user input devices, and provide information to the programmer.

2.3 DATA TYPES

We have described the data stores which are available to the programmer; what sort of data can he keep in these stores? A variety of data types exist, each of which is useful for solving certain types of problems. These data types are divided into two classes: numeric and string.

The operations which can be performed on these data types are described beginning in Chapter4.

2.3.1 NUMERIC DATA

2.3.1.1 INTEGERS

Integers are the simplest form of numeric data. They consist of some number of bits (commonly 16) representing a base two integer number.

Integers used for performing arithmetic need to be signed. Hence they are stored in two's-complement binary form, with the high-order bit indicating the sign. A sign bit of 0 signifies a non-negative number, while that of 1 signifies a negative one. Some integers and their decimal equivalents are:

```
000...000 =  0 (base 10)
000...001 =  1
111...111 = -1
111...100 = -4
```

The value of an n-bit signed integer ranges from -2**(n-1) up to +2**(n-1) -1. Thus a halfword integer can have the values -32,768 up to +32,767.

It is also possible to work with unsigned integers, which are called logical. The value of a n-bit logical integer ranges from 0 up to +2**n -1. The most important use of logical integers is for addressing data stores, which have locations numbered from 0 on up. All references to stores must eventually generate a logical integer to act as the final location address.

2.3.1.2 FRACTIONS

It is possible for the programmer to work with signed fractions on the META 4B. A fraction is represented as an n-bit two's-complement binary number, the high-order bit indicating the sign. The binary point is assumed to lie between the sign bit and the next high-order bit. This allows fractions in the range -1.0 up to +.999... Examples of fractions are:

```
010...000 =  .5 (base 10)
011...000 =  .75
011...111 =  .999...
110...000 = -.5
100...000 = -1.0
```

Fractions are a necessary data type on the META 4B because the VG display scope uses fractional coordinates.

2.3.1.3 INDEX-BASE-DISPLACEMENT

An index-base-displacement (XBD) is a data type which is present only within instructions. It is used to generate an integer which can be used for various purposes during the execution of the instruction. Typically this integer is treated logically and used as a main store address in order to obtain one or more bytes for the instruction to operate upon.

The integer is generated from the sum of three other integers specified by the index, base, and displacement. The base is a 4-bit field (called the B field in the instruction) which specifies one of the sixteen general-purpose registers, the contents of which is treated as an integer. Added to this integer is the contents of the GPR specified by the 4-bit index (X) field. Finally, the displacement, a 12-bit logical integer present immediately within the instruction (in the D field) is added. If the base or index field specifies GPR0, that field is ignored and GPR0 is not added to the sum.

If the XBD is to be used as an address, the above computation will produce the expected result regardless of whether the X, B, or D components are considered to be signed or logical integers; the final sum is a logical integer specifying a location in main store.

2.3.1.4 PROCEDURE DISPLACEMENT

See Chapter 3.1.1 for an explanation.

2.3.2 STRING DATA

2.3.2.1 BYTE

A byte is the simplest form of string data. It consists of any single byte of information, which could be an 8-bit number, a character, or a flag. A byte is also a character string of length one (see next paragraph).

2.3.2.2 CHARACTER STRING

A character string is a sequence of bytes in main store, having a length from zero (the null string) to 65,535. Character strings are used to represent arbitrary length logical data (e.g., a PL/I bit string) or character strings in the usual sense (e.g., messages).

3 PROGRAM FACILITIES

The q-interpreter provides a comprehensive set of facilities which can be used directly in assembly language or via a high-level language. Specifications for a high-level language such as ALGOL W will hide many of the q-interpreter features from the programmer and hopefully make it easier to program. However, it is the purpose of this document to present all of the features for posterity; hence we will be oriented toward the assembly language programmer.

It is assumed that the reader is familiar with the META 4A Principles of Operation and the facilities provided by Waterloo Assembler G (ASMG). The differences between S/360 ASMG and BUGSASM A, which are outlined in the META 4A Assembler Users' Guide, do not hold for the META 4B, however. Any differences will be presented in this document. The general format used herein is to present each META 4B feature and its use in conjunction with BUGSASM B.

The metalinguistic symbols used in this document to specify syntax are described in Appendix 0.

3.1 PROCEDURE DESCRIPTION

3.1.1 INTRODUCTION

The average programmer is accustomed to thinking of his program as a sequence of individual computations leading to the solution of his problem. Programming practice dictates that these computations should be grouped into logical sets of associated computations, each of which performs a specific portion of the overall job. The META 4B supports such a concept by requiring a program to be split up into procedures. Sequencing of procedures is controlled by procedure call and return. During the execution of one procedure, other procedures may be called (either explicitly by the programmer or implicitly by the q-interpreter), execute, and return.

The q-interpreter is at all times executing a specific procedure, called the current procedure. The execution of this procedure is maintained by three registers in the register file. The first, control register 16, is called the Procedure Base Register (PBR). It simply contains the main store address of the beginning of the current procedure (remember, main store is addressed in bytes!). Since a procedure must begin on a halfword boundary, the

PBR is always forced to be even (i.e., the low-order bit is ignored). In addition, whenever a new procedure is entered, GPR15 in the register file is set to contain a copy of the PBR. This is useful for implicit addressing of static data in the procedure, and should not be modified by the programmer.

PBR

```
┌──────────────────┐
| proc. address 0|
└──────────────────┘
 0                15
```

The third register is the Procedure Displacement Register (PDR), control register 17. It contains the byte displacement from the PBR to the next instruction to be executed. Whenever the q-interpreter is ready to execute an instruction, it fetches it from the locations specified by the sum of the PBR and PDR, and then adjusts the PDR so as to be ready for the next instruction.

Procedures are limited to 4K bytes in length. Furthermore, instructions must be on halfword boundaries, so the low-order bit of the PDR is ignored, as with the PBR.

PDR

```
┌──────────────────┐
|0000proc. disp.0|
└──────────────────┘
 0   4            15
```

Procedure displacements are a standard data type, and can reside in areas other than the PDR. For example, instructions which alter the normal sequential execution path (branching instructions) contain such displacements.

3.1.2 PROGRAMMING

A single assembly may contain any number of procedures, each of which is coded as follows:

```
[EXTERNAL] PROC <name>
               •
               •
               •  (procedure code)
               •
               •
          static data
```

The PROC statement specifies the start of a new procedure with the name indicated by <name>. Each procedure must have an identifying name. If the scope of the procedure is to be external, that is, if the procedure name can be referenced by other assemblies (e.g., via V-constants), the specification EXTERNAL must be included.

Following the PROC statement is up to 4K bytes of code and data which performs the computations assigned to this procedure. Static data not used by other procedures should be placed at the end, including a LTORG statement to locate all literals. The PROC statement sets up a USING on GPR15 so that this data can be implicitly addressed.

NOTE that it is not necessary to code a CSECT statement anywhere in the assembly.

3.2 INSTRUCTION DESCRIPTION

3.2.1 INTRODUCTION

Instructions on the MET 4B are similar to those on an IBM System/360/370[ ]. Each instruction consists of an operation, which specifies some function to be performed upon one or two <u>operands</u>.

The operation is specified in an instruction by a four-, eight-, or twelve-bit operation code. This code specifies not only the basic function to be performed, but also certain modifiers, such as the data type of the operands. Each different operation code is given a mnemonic for use in assembly language programming (e.g., "A" for add).

Operations can be performed upon single operands (<u>unary</u> operations), or upon two operands (<u>binary</u> operations). "Reference codes" are appended to an operation mnemonic to specify the data type and location of the operands (e.g., "ARR" for adding the contents of two general-purpose registers). The next section describes the various types of operands and their reference codes.

When programming in assembly language, instruction operands are specified in their logical order. The instruction descriptions in the document give the symbolic format for these operands. A general-purpose register is shown symbolically as an "R", while an XBD address is shown symbolically as "D(X,B)". A symbolic operand may have a "1" or a "2" suffix to denote which operand it is, or an "S" or "D" suffix to denote source and destination. Thus an instruction to add two registers is shown as:

```
ARR     R1,R2
```

3.2.2 OPERANDS

Various restrictions are placed upon data types if they are to be operated upon directly by an instruction. These restrictions are outlined here:

3.2.2.1 INTEGERS

In most cases, integers must be sixteen bits in length in order to be operated upon by instructions. In a few cases they must be 32 bits long, such as for the dividend in a divide operation. These integers may reside in the general-purpose registers (and if so are given the reference code "R"), within instructions as immediate data (code "I"), or within a halfword in main store (code "H").

3.2.2.2 FRACTIONS

The restriction placed upon integers also hold for fractions.

3.2.2.3 INDEX-BASE-DISPLACEMENTS

XBD data can only reside within instructions, and are given the code "A". They consist of a four-bit X field, a four-bit B field, and a twelve-bit D field.

3.2.2.4 PROCEDURE DISPLACEMENTS

Procedure displacements typically reside within the PDR control register or branching instructions, but they can also be stored in GPRs (code "R") or halfwords in main store (code "H"). Since they require only twelve bits, the high-order four bits of the register or halfword are ignored and assumed to be zero.

3.2.2.5 BYTES

Single byte data items can reside in the low-order eight bits of a GPR (code "R"), an instruction (code "I"), or in any byte in main store (code "B"). When in main store, bytes do not necessarily have to be on halfword boundaries.

3.2.2.6 CHARACTER STRINGS

Character strings can only reside in main store and are given the code "C". They may begin on any byte boundary and be of any length from 0 (the *null* string) up to 65,535 bytes.

3.3 PROCEDURE CHECKS

Certain errors can arise during the execution of a program, such as an attempt to divide by zero. The q-interpreter provides a means of informing the monitor and/or user of these errors, a means called *procedure checks*. When an error is detected, execution of the instruction in question is aborted, and an implicit procedure call occurs. A detailed explanation is given in Chapter 12.

With each instruction description in the following chapters is given a list of the possible procedure checks that can occur and why.

4 THE TRANSFER DATA INSTRUCTION

The purpose of this chapter is to serve as an introduction to the instruction set of the META 4B by describing a fundamental instruction, XFER. NOTE that the format used to describe this instruction will be used throughout the remaining chapters.

Each instruction description consists of four parts:

1. The name of the instruction.

2. The mnemonic used when coding the instruction, followed by the symbolic format of the operands.

3. A picture of the instruction as it resides in main store, with its operand code (hexadecimal) and operand fields. Unused bits are indicated by a slash.

4. An English-language description of the instruction.

transFER data

```
XFER      D,DD(BD),S,DS(BS),DN(BN)
+---------+-----+-----+-----+---------+-----+---------+-----+--------+
|   FF    |/ D  |/ S  | BD  |  DD  DS | BS  |         | BN  |   DN   |
+---------+-----+-----+-----+---------+-----+---------+-----+--------+
0         8     12    16    20        32    36        48    52      63
```

The XFER instruction allows the programmer to transfer one or more halfwords of data from locations in one store to locations in the same or any other store. Data is transferred from the source store specified by S, starting at the location specified by DS(BS), to the destination store specified by D, starting at the location DD(BD). The number of halfwords transferred is specified by DN(BN).

Addresses and the length are computed by adding the contents of the base GPR (BD, BS, or BN) to the immediate twelve-bit displacement (DD, DS, or DN), forming a logical integer. If a base field contains zero, GPR0 is not added; the displacement is used by itself. The reader may have noticed that the addresses and length are XBD data types without the index.

The stores which can be specifed in the three-bit D or S fields and their idiosyncracies are as follows:

code 0: Register file (R). Addresses are treated modulo 64, so that transferring wraps from register 63 to 0.

code 1: Local Store (LS). Addresses are treated modulo n, n being the current size of local store. Thus transferring wraps from location n-1 to 0.

code 2: Main Store (MS). The address must specify a halfword boundary. If it does not, an alignment procedure check occurs.

code 3: Vector General Register file (VGR). Addresses are treated modulo 128, so that transferring wraps from register 127 to 0. See chapter 15.2. for an explanation of the Vector General registers.

codes 4-7: unused. If specified as a source, zeroes are obtained; if as a destination, the data falls off the face of the earth.

If the number of halfwords to be transferred is zero, you wouldn't believe what happens.

In order to simplify the specification of store types, registers, etc., a macro is provided which generates equates for them. This macro is called "M4BEQUS" and should be included at the end of all META 4B assemblies. A listing of the generated code can be found in Appendix 1.

Examples:

```
XFER      R,R5,MS,PLACE,3
```

Three halfwords are transferred from main store, starting at PLACE, into GPR5-7.

```
XFER      LS,256,LS,128,128
```

128 halfwords are transferred from local store locations 128-255 to locations 256-383.

```
XFER      LS,0(R2),VGR,VGDIAL1,0(R3)
```

The number of Vector General dial registers specified by the contents of GPR3 is transferred into local store starting at the location specified by the contents of GPR2.

```
XFER      R,R8,R,DBR,1
```

The DBR is placed into GPR8.

NOTE that XFER is not intended for transferring single data items among the general-purpose registers and main store. There are other, more powerful, instructions for this purpose, which are described in later chapters.

5 BRANCHING INSTRUCTIONS

5.1 INTRODUCTION

A class of operations known as branching operations are provided to allow the programmer to make decisions and alter the flow of control through his procedures. Branching decisions are controlled by the Condition Flag Register.

5.2 CONDITION FLAG REGISTER

The CFR, control register 18, provides the means for making decisions in a procedure. It contains eight condition bits which are set by certain instructions in order to inform the programmer of the results of the instruction. For example, compare operations set the CFR to reflect whether the first operand was less than, equal to, or greater than the second operand.

```
CFR
┌────────┬─┬────────┐
| flags  |S|00000000|
└────────┴─┴────────┘
 0      7 8        15
```

Whenever the CFR is modified by an instruction, all bits are initially set to zero. Next, one of the flags (bits 0-6) is set to reflect the results of the operation. (In the case of compares, bit 0 is set on if the operands are equal, bit 1 if the first operand is greater, or bit 2 if it is less than the second operand.) Finally, the summary flag, bit 7, is set to reflect the most important condition. (For compares, it is set off if the operands are unequal, or on if they are equal.)

The purpose of some branching instructions is to test the CFR and either branch or not, depending upon the test.

5.3 UNCONDITIONAL BRANCHING INSTRUCTIONS

No OPeration
NOP   0

```
┌─────────┬─────────┐
|    OD   |00000000 |
└─────────┴─────────┘
 0         8       15
```

NOP performs no operation whatsoever. Execution continues with the next sequential intruction.

Branch
B   label

```
┌────┬──────────────┐
|  4 |proc. disp.0  |
└────┴──────────────┘
 0    4            15
```

A branch is taken to the specified label regardless of the setting of the CFR. Such a branch is called <u>unconditional</u>.

Branch via Register
BR   R

```
┌──────────────┬────┐
|     OFC      | R  |
└──────────────┴────┘
 0             12 15
```

Branch via Halfword   [Note the mnemonic]
BH   D(X,B)

```
┌──────────────┬────┬────┬────────────────┐
|     9FC      | X  | B  |       D        |
└──────────────┴────┴────┴────────────────┘
 0             12   16   20             31
```

An unconditional branch is taken. For BR, the procedure displacement is obtained from the GPR specifed by the operand. For BH, it is obtained from the halfword at the main store address specified. In both cases, the high-order four bits of the operand are ignored.

An alignment procedure check occurs during BH if the main store address is odd.

5.4 CONDITIONAL BRANCHING INSTRUCTIONS

Branch True
BT   label

```
┌────┬──────────────┐
│  3 │proc. disp.0│
└────┴──────────────┘
 0    4            15
```

Branch False
BF   label

```
┌────┬──────────────┐
│  2 │proc. disp.0│
└────┴──────────────┘
 0    4            15
```

If the summary flag is on and the operation is BT, or if it is off and the operation is BF, and branch is taken to the specifed label. Otherwise no branch is taken.

Branch on Condition Flag register
BCF   mask,label

```
┌─────────┬─────────┬─────────────────┐
│    5D   │  mask   │0000proc. disp.0│
└─────────┴─────────┴─────────────────┘
 0         8         16              31
```

The eight-bit mask is used to select bits in the CFR. If <u>any</u> selected bits are on, a branch is taken to the label. Otherwise, no branch is taken. Each bit in the mask corresponds to a bit in the CFR. Wherever a one bit appears in the mask, the corresponding CFR bit is selected for testing.

It is not usually necessary for the programmer to specify a mask on a BCF instruction. Instead, "extended mnemonics" are provided which allow the user to ignore the mask. Examples are BE (Branch Equal) and BNG (Branch Not Greater). Extended mnemonics are described with the relevant instructions, and are also listed in Appendix 2a.

5.5 CASE

```
CASE      R,D(X,B)

┌──────────┬──────┬──────┬──────┬─────────────────┐
|    6D    |  R   |  X   |  B   |        D        |
└──────────┴──────┴──────┴──────┴─────────────────┘
0          8      12     16     20               31
```

The main store address specifies a table of halfwords containing procedure displacements. The n'th halfword is selected, and a branch is taken to the procedure displacement within this halfword. As usual, the high-order four bits are ignored. The value of n is the logical integer in the GPR specified by R, and ranges frcm 0 on up.

An alignment procedure check cccurs if the table address is odd.

In order to simplify the generation of CASE tables, the DCPD (Define Constant Procedure Displacements) statement is provided. It is coded as follows:

```
[label] DCPD    label1,label2,...,labeln
```

A table of n procedure displacements is generated.

6 DATA MOVING INSTRUCTIONS

6.1 UNARY INSTRUCTIONS

Set to Zero Register
SZR R

```
┌────────────────┬─────┐
│      0F0       │  R  │
└────────────────┴─────┘
 0               12  15
```

Set to Zero Halfword
SZH D(X,B)

```
┌────────────────┬─────┬─────┬────────────────┐
│      9F0       │  X  │  B  │       D        │
└────────────────┴─────┴─────┴────────────────┘
 0               12    16    20               31
```

These two instructions cause their operand to be set to zero. For SZR, the specified GPR is set to zero, while for SZH, the halfword at the main store address is zeroed.

An alignment procedure check occurs during SZH if the address is odd.

Set to One Register
SOR R

```
┌────────────────┬─────┐
│      0F1       │  R  │
└────────────────┴─────┘
 0               12  15
```

Set to One Halfword
SOH D(X,B)

```
┌────────────────┬─────┬─────┬────────────────┐
│      9F1       │  X  │  B  │       D        │
└────────────────┴─────┴─────┴────────────────┘
 0               12    16    20               31
```

These two instructions cause their operand to be set to the integer 1. For SOR, the specifed GPR is set to 1, while for SOH, the main store halfword is set to 1.

An alignment procedure check cccurs during SOH if the address is odd.

6.2 REPLACE

Replace Register with Register
RRR   R1,R2

```
┌─────────┬─────┬─────┐
|   06    | R1  | R2  |
└─────────┴─────┴─────┘
 0         8     12  15
```

Replace Register with Immediate
RRI   R1,IH2

```
┌─────────┬─────┬─────┬───────────────────┐
|   56    | R1  |/////|        IH2        |
└─────────┴─────┴─────┴───────────────────┘
 0         8     12    16                31
```

Replace Register with Halfword
RRH   R1,D2(X2,B2)

```
┌─────────┬─────┬─────┬─────┬─────────────┐
|   66    | R1  | X2  | B2  |     D2      |
└─────────┴─────┴─────┴─────┴─────────────┘
 0         8     12    16    20          31
```

Replace Register with Address
RRA   R1,D2(X2,B2)

```
┌─────────┬─────┬─────┬─────┬─────────────┐
|   76    | R1  | X2  | B2  |     D2      |
└─────────┴─────┴─────┴─────┴─────────────┘
 0         8     12    16    20          31
```

Replace Register with Byte
RRB   R1,D2(X2,B2)

```
┌─────────┬─────┬─────┬─────┬─────────────┐
|   86    | R1  | X2  | B2  |     D2      |
└─────────┴─────┴─────┴─────┴─────────────┘
 0         8     12    16    20          31
```

Replace Halfword with Register
RHR   D1(X1,B1),R2

```
┌─────────┬─────┬─────┬─────┬─────────────┐
|   96    | R2  | X1  | B1  |     D1      |
└─────────┴─────┴─────┴─────┴─────────────┘
 0         8     12    16    20          31
```

DEST	SOURCE
R	R
H	H
B	B
C	I
	A
	C

RR	HR	BR	CC
RI	HI	BI	
RH	HH		
RA	HA		
RB			

Replace Byte with Register
RBR D1(X1,B1),R2

```
┌────────┬────┬────┬────┬────────────┐
|   A6   | R2 | X1 | B1 |     D1     |
└────────┴────┴────┴────┴────────────┘
0        8    12   16   20          31
```

Replace Halfword with Immediate
RHI D1(X1,B1),IH2

```
┌────────┬────┬────┬────┬────────────┬────────────────┐
|   B6   |////| X1 | B1 |     D1     |      IH2       |
└────────┴────┴────┴────┴────────────┴────────────────┘
0        8    12   16   20           32              47
```

Replace Halfword with Address
RHA D1(X1,B1),D2(X2,B2)

```
┌────────┬────┬────┬────┬────────────┬────┬────────────┐
|   C6   | X2 | X1 | B1 |     D1     | B2 |     D2     |
└────────┴────┴────┴────┴────────────┴────┴────────────┘
0        8    12   16   20           32   36          47
```

Replace Halfword with Halfword
RHH D1(X1,B1),D2(X2,B2)

```
┌────────┬────┬────┬────┬────────────┬────┬────────────┐
|   D6   | X2 | X1 | B1 |     D1     | B2 |     D2     |
└────────┴────┴────┴────┴────────────┴────┴────────────┘
0        8    12   16   20           32   36          47
```

Replace Byte with Immediate
RBI D1(X1,B1),IB2

```
┌────────┬────┬────┬────┬────────────┬─────────┬────────┐
|   E6   |////| X1 | B1 |     D1     |/////////|  IB2   |
└────────┴────┴────┴────┴────────────┴─────────┴────────┘
0        8    12   16   20           32        40      47
```

Replace Character string with Character string
RCC D1(X1,B1),D2(X2,B2),DL(BL)

```
┌────────┬────┬────┬────┬────────────┬────┬────────────┬────┬────────────┐
|   F6   |X2  |X1  |B1  | D1         |B2  | D2         |BL  |     DL     |
└────────┴────┴────┴────┴────────────┴────┴────────────┴────┴────────────┘
0        8    12   16   20           32   36           48   52          63
```

These instructions cause the first operand to be replaced by the second operand.

LR R1,R2 RRR causes the GPR specified by R1 to contain the contents of the GPR specified by R2. RRI causes the GPR specified by R1 to contain te immediate halfword. Immediate data may be specified by any self-defining expression.

L R1,ADDR RRH causes the GPR specified by R1 to contain the halfword at the specified main store address.

LA R1,ADDR RRA causes the R1 GPR to contain the value of the specified XBD address. The address itself is the second operand -- no main store data is used.

LB RRB causes the low-order eight bits of the R1 GPR to contain the byte at the main store address. The high-order eight bits of the GPR are zeroed.

ST RHR causes the halfword at the main store address to contain the GPR specified by R2.

STB RBR causes the byte at the main store address to contain the low-order eight bits of the GPR specified by R2.

MVHI RHI causes the specifed main store halfword to contain the immediate halfword.

MVA RHA causes the specifed main store halfword (first operand) to contain the second operand XBD address. This address is not used to reference main store, but is itself the second operand.

MVH RHH causes the first operand main store halfword to contain the second operand main store halfword.

MVBI RBI causes the byte at the main store address to contain the immediate byte.

MVC RCC causes a copy of the second operand character string to be placed into the first operand string. This copy is made by logically lifting the string cut of main store and setting it down in the first operand, so that no propagation occurs. The length of the two strings is specifed by DL(BL), and is computed as follows: the contents of the GPR specified by BL (unless it is zero) is added to the twelve-bit displacement DL.

An alignment procedure check can occur on any instruction requiring a halfword in main store if its address is odd.

6.3 SWAP

RR
RH = HR
RB = BR
HH

SWap Register with Register
SWRR   R1,R2

```
+----------+-----+-----+
|    07    | R1  | R2  |
+----------+-----+-----+
0          8     12  15
```

SWap Register with Halfword
SWRH   R1,D2(X2,B2)

```
+----------+-----+-----+-----+--------------+
|    67    | R1  | X2  | B2  |      D2      |
+----------+-----+-----+-----+--------------+
0          8     12    16    20            31
```

SWap Register with Byte
SWRB   R1,D2(X2,B2)

```
+----------+-----+-----+-----+--------------+
|    87    | R1  | X2  | B2  |      D2      |
+----------+-----+-----+-----+--------------+
0          8     12    16    20            31
```

SWap Halfword with Register
SWHR   D1(X1,B1),R2

```
+----------+-----+-----+-----+--------------+
|    97    | R2  | X1  | B1  |      D1      |
+----------+-----+-----+-----+--------------+
0          8     12    16    20            31
```

SWap Byte with Register
SWBR   D1(X1,B1),R2

```
+----------+-----+-----+-----+--------------+
|    A7    | R2  | X1  | B1  |      D1      |
+----------+-----+-----+-----+--------------+
0          8     12    16    20            31
```

SWap Halfword with Halfword
SWHH   D1(X1,B1),D2(X2,B2)

```
+----------+-----+-----+-----+--------------+-----+--------------+
|    D7    | X2  | X1  | B1  |      D1      | B2  |      D2      |
+----------+-----+-----+-----+--------------+-----+--------------+
0          8     12    16    20             32    36            47
```

SWap Character string with Character string
SWCC D1(X1,B1),D2(X2,B2),DL(BL)
(macro)

The swap operation is used to interchange the contents of the two instruction operands.

SWRR causes the contents of the two specifed GPRs to be swapped.

SWRH causes the contents of the GPR and the main store halfword to be swapped.

SWRB causes the low-order eight bits of the GPR to be swapped with the byte in main store. The high-order eight bits of the GPR are set to zero.

SWHR performs the same function as SWRH.

SWBR performs the same function as SWRB.

SWHH causes the contents of the two main store halfwords to be swapped.

SWCC causes the two character strings to be interchanged. The length of the strings is specified by DL(BL). As with all character string operations, no propagation occurs. SWCC is not implemented in the q-interpreter, but rather by a macro which generates three XCC instructions.

An alignment procedure check will occur on instructions which specify a main store halfword not on an even boundary.

7 SHIFT INSTRUCTIONS

7.1 INTRODUCTION

The shift instructions allow the programmer to shift the contents of a 16- or 32-bit value in the GPRs. The first operand is always a GPR number specifying the GPR(s) whose contents are to be shifted. The second operand specifies the shift count (number of bit positions to be shifted), which can be located immediately within the instruction or in a GPR. Immediate counts are four bits long, allowing a value from zero to fifteen. If the count is in a GPR, the low-order five bits are used, allowing a value from zero to 31.

7.2 ARITHMETIC SHIFT INSTRUCTIONS

Left SHift, Immediate
LSHI R1,IX2

```
┌─────────┬─────┬─────┐
│   14    │ R1  │IX2  │
└─────────┴─────┴─────┘
 0        8     12  15
```

Left SHift, Register
LSHR R1,R2

```
┌─────────┬─────┬─────┐
│   1C    │ R1  │ R2  │
└─────────┴─────┴─────┘
 0        8     12  15
```

Right SHift, Immediate
RSHI R1,IX2

```
┌─────────┬─────┬─────┐
│   15    │ R1  │IX2  │
└─────────┴─────┴─────┘
 0        8     12  15
```

Right SHift, Register
RSHR R1,R2

```
┌──────────┬──────┬──────┐
|    1D    |  R1  |  R2  |
└──────────┴──────┴──────┘
 0          8      12  15
```

The contents of the GPR specified by R1 is shifted. The direction of shift is specified by the op code: left shifts cause vacated bit positions to be filled with zeroes; right shifts cause them to be filled with the original sign bit (original bit 0). The shift count is either the immediate field IX2 or the low-order five bits of the GPR specified by R2.

NOTE that these shifts can be used to multiply or divide the GPR by a power of two.

If, during left shifts, a bit unlike the original sign bit is shifted into bit 0, overflow is considered to have occurred, because a significant bit has been shifted out of the register. Such a condition is reflected in the CFR: flag bit 1 is set off if it does not occur or on if it does (the summary flag is always off). The extended mnemonic BO (Branch on Overflow) can be used to test this condition. NOTE that it is impossible to branch on no overflow; overflow is the one exception to the CFR-setting rules described in Chapter 5.2.

Left SHift Double, Immediate
LSHDI R1,IX2

```
┌──────────┬──────┬──────┐
|    16    |  R1  | IX2  |
└──────────┴──────┴──────┘
 0          8      12  15
```

Left SHift Double, Register
LSHDR R1,R2

```
┌──────────┬──────┬──────┐
|    1E    |  R1  |  R2  |
└──────────┴──────┴──────┘
 0          8      12  15
```

Right SHift Double, Immediate
RSHDI R1,IX2

```
┌──────────┬──────┬─────┐
|    17    |  R1  | IX2 |
└──────────┴──────┴─────┘
 0          8      12  15
```

Right SHift Double, Register
RSHDR R1,R2

```
┌──────────┬──────┬─────┐
|    1F    |  R1  | R2  |
└──────────┴──────┴─────┘
 0          8      12  15
```

These instructions operate exactly as the four explained above, except that a 32-bit operand (GPR pair) is shifted. The low-order sixteen bits of the operand are in the GPR specified by R1, while the high-order sixteen bits are in the previous (I said previous) GPR (i.e., R1-1).

A register specification procedure check occurs if R1 is zero, since no previous GPR exists (GPR-1?)

7.3 LOGICAL SHIFT INSTRUCTIONS

Left SHift Logical, Immediate
LSHLI   R1,IX2

```
┌──────────┬─────┬─────┐
│    10    │ R1  │ IX2 │
└──────────┴─────┴─────┘
0          8     12  15
```

Left SHift Logical, Register
LSHLR   R1,R2

```
┌──────────┬─────┬─────┐
│    18    │ R1  │ R2  │
└──────────┴─────┴─────┘
0          8     12  15
```

Right SHift Logical, Immediate
RSHLI   R1,IX2

```
┌──────────┬─────┬─────┐
│    11    │ R1  │ IX2 │
└──────────┴─────┴─────┘
0          8     12  15
```

Right SHift Logical, Register
RSHLR   R1,R2

```
┌──────────┬─────┬─────┐
│    19    │ R1  │ R2  │
└──────────┴─────┴─────┘
0          8     12  15
```

The contents of the GPR specified by R1 is shifted. The direction of shift is specified by the operation code: both directions cause vacated bit positions to be filled with zeroes. The shift count is either the immediate field IX2 or the low-order five bits of the GPR specified by R2.

The CFR is unchanged.

Left SHift Logical Double, Immediate
LSHLDI   R1,IX2

```
┌─────────┬─────┬─────┐
│   12    │ R1  │IX2  │
└─────────┴─────┴─────┘
 0         8    12  15
```

Left SHift Logical Double, Register
LSHLDR   R1,R2

```
┌─────────┬─────┬─────┐
│   1A    │ R1  │ R2  │
└─────────┴─────┴─────┘
 0         8    12  15
```

Right SHift Logical Double, Immediate
RSHLDI   R1,IX2

```
┌─────────┬─────┬─────┐
│   13    │ R1  │IX2  │
└─────────┴─────┴─────┘
 0         8    12  15
```

Right SHift Logical Double, Register
RSHLDR   R1,R2

```
┌─────────┬─────┬─────┐
│   1B    │ R1  │ R2  │
└─────────┴─────┴─────┘
 0         8    12  15
```

These instructions operate exactly as the four explained above, except that a 32-bit operand (GPR pair) is shifted. The low-order sixteen bits of the operand are in the GPR specified by R1, while the high-order sixteen bits are in the previous GPR (i.e., R1-1).

A register specification procedure check occurs if R1 is zero, since no previous GPR exists.

8 ARITHMETIC INSTRUCTIONS

8.1 TEST SIGN

Test Sign Register
TSR R

```
┌────────────┬────┐
|    0F8     | R  |
└────────────┴────┘
0            12  15
```

Test Sign Address
TSA D(X,B)

```
┌────────────┬────┬────┬────────────┐
|    7F8     | X  | B  |     D      |
└────────────┴────┴────┴────────────┘
0            12   16   20          31
```

Test Sign Halfword
TSH D(X,B)

```
┌────────────┬────┬────┬────────────┐
|    9F8     | X  | B  |     D      |
└────────────┴────┴────┴────────────┘
0            12   16   20          31
```

The operand is treated as a signed integer and its sign is tested. The CFR is set as follows: bit 0 is set if the operand is zero; bit 1 if it is positive; or bit 2 if it is negative. The summary flag is set to 0 if the operand is zero, or 1 otherwise.

The operand for TSR is the specified GPR. For TSA it is the XBD address itself. And for TSH it is the halfword at the specified main store address.

Extended mnemonics are provided for use with the BCF instruction. They are: BZ (Branch Zero), BNZ (Branch Not Zero), BP (Branch Positive), BNP (Branch Not Positive), BN (Branch Negative), and BNN (Branch Not Negative).

An alignment procedure check cccurs during TSH if the halfword address is odd.

8.2 UNARY INCREMENT

Increment Register
IR R

```
┌──────────────┬─────┐
│     0F2      │  R  │
└──────────────┴─────┘
 0             12  15
```

Increment Halfword
IH D(X,B)

```
┌──────────────┬────┬────┬──────────────┐
│     9F2      │ X  │ B  │      D       │
└──────────────┴────┴────┴──────────────┘
 0             12   16   20           31
```

Increment Increment Register
IIR R

```
┌──────────────┬─────┐
│     0F3      │  R  │
└──────────────┴─────┘
 0             12  15
```

Increment Increment Halfword
IIH D(X,B)

```
┌──────────────┬────┬────┬──────────────┐
│     9F3      │ X  │ B  │      D       │
└──────────────┴────┴────┴──────────────┘
 0             12   16   20           31
```

The operand is incremented by one (IR, IH) or two (IIR, IIH), depending upon the operation code.

The operand for IR or IIR is the specified GPR, while for IH or IIH it is the halfword at the main store address. An alignment procedure check occurs if this address is odd.

8.3 UNARY DECREMENT AND TEST

Decrement, Test Sign Registe
DTSR R

```
┌──────────────┬─────┐
|     0F4      |  R  |
└──────────────┴─────┘
0              12  15
```

Decrement, Test Sign Halfword
DTSH D(X,B)

```
┌──────────────┬────┬────┬──────────────┐
|     9F4      | X  | B  |      D       |
└──────────────┴────┴────┴──────────────┘
0              12   16   20            31
```

Decrement Decrement, Test Sign Register
DDTSR R

```
┌──────────────┬─────┐
|     0F5      |  R  |
└──────────────┴─────┘
0              12  15
```

Decrement Decrement, Test Sign Halfword
DDTSH D(X,B)

```
┌──────────────┬────┬────┬──────────────┐
|     9F5      | X  | B  |      D       |
└──────────────┴────┴────┴──────────────┘
0              12   16   20            31
```

The operand is decremented by one (DTSR, DTSH) or two (DDTSR, DDTSH), depending upon the operation code. Following this, the sign of the resulting number is tested, and the CFR is set as for the Test Sign instructions described above.
The operand for DTSR or DDTSR is the specified GPR, while for DTSH and DDTSH it is the halfword at the main store address. An alignment procedure check occurs if this address is odd.

8.4 ABSOLUTE VALUE AND NEGATE

Absolute value Register
ABSR R

```
┌──────────────┬─────┐
│     0F6      │  R  │
└──────────────┴─────┘
0             12  15
```

Absolute value Halfword
ABSH D(X,B)

```
┌──────────────┬─────┬─────┬───────────────┐
│     9F6      │  X  │  B  │       D       │
└──────────────┴─────┴─────┴───────────────┘
0             12    16    20              31
```

The operand is treated as a signed number and replaced by its absolute value. The absolute value of the maximum negative number is again the maximum negative number.

The operand for ABSR is the specified GPR, while for ABSH it is the halfword at the main store address. An alignment procedure check occurs if the address is odd.

Negate Register
NEGR R

```
┌──────────────┬─────┐
│     0F7      │  R  │
└──────────────┴─────┘
0             12  15
```

Negate Halfword
NEGH D(X,B)

```
┌──────────────┬─────┬─────┬───────────────┐
│     9F7      │  X  │  B  │       D       │
└──────────────┴─────┴─────┴───────────────┘
0             12    16    20              31
```

The operand is treated as a signed number and replaced by its negative. The negative of zerc is again zero. The negative of the maximum negative number is again the maximum negative number.

The operand for NEGR is the specified GPR, while for NEGH it is the halfword at the main store address. An alignment procedure check occurs if the address is odd.

8.5 ADD

RR	HR
RI	HI
RH	HH
RA	HA

Add Register plus Register
ARR R1,R2

```
┌─────────┬─────┬─────┐
|   01    | R1  | R2  |
└─────────┴─────┴─────┘
 0         8    12  15
```

Add Register plus Immediate
ARI R1D,R1S,IH2 -or- R1,IH2

```
┌─────────┬─────┬─────┬────────────────┐
|   51    |R1D  |R1S  |      IH2       |
└─────────┴─────┴─────┴────────────────┘
 0         8    12    16              31
```

Add Register plus Halfword
ARH R1,D2(X2,B2)

```
┌─────────┬─────┬─────┬─────┬────────────┐
|   61    | R1  | X2  | B2  |     D2     |
└─────────┴─────┴─────┴─────┴────────────┘
 0         8    12    16    20          31
```

Add Register plus Address
ARA R1,D2(X2,B2)

```
┌─────────┬─────┬─────┬─────┬────────────┐
|   71    | R1  | X2  | B2  |     D2     |
└─────────┴─────┴─────┴─────┴────────────┘
 0         8    12    16    20          31
```

Add Halfword plus Register
AHR D1(X1,B1),R2

```
┌─────────┬─────┬─────┬─────┬────────────┐
|   91    | R2  | X1  | B1  |     D1     |
└─────────┴─────┴─────┴─────┴────────────┘
 0         8    12    16    20          31
```

Add Halfword plus Immediate
AHI D1(X1,B1),IH2

```
┌─────────┬─────┬─────┬─────┬────────────┬────────────────┐
|   B1    |/////| X1  | B1  |     D1     |      IH2       |
└─────────┴─────┴─────┴─────┴────────────┴────────────────┘
 0         8    12    16    20           32              47
```

Add Halfword plus Address

AHA   D1(X1,B1),D2(X2,B2)

```
┌───────────┬─────┬─────┬─────┬─────────────────┬─────┬───────────────┐
│    C1     │ X2  │ X1  │ B1  │       D1        │ B2  │      D2       │
└───────────┴─────┴─────┴─────┴─────────────────┴─────┴───────────────┘
0           8     12    16    20                32    36             47
```

Add Halfword plus Halfword

AHH   D1(X1,B1),D2(X2,B2)

```
┌───────────┬─────┬─────┬─────┬─────────────────┬─────┬───────────────┐
│    D1     │ X2  │ X1  │ B1  │       D1        │ B2  │      D2       │
└───────────┴─────┴─────┴─────┴─────────────────┴─────┴───────────────┘
0           8     12    16    20                32    36             47
```

The two operands are added together, and the first operand is replaced by the sum. Carry out of the sign bit is recorded in bit 0 of the CFR, while overflow (the 'exclusive or' of the carries out of the sign bit and bit 1) is recorded in bit 1 of the CFR. The summary flag is always zero.

In addition to the BO extended mnemonic already described, there exists BC (Branch Carry). Remember, it is impossible to branch on no carry or no overflow. The operands for ARR are the two specified GPRs.

For ARI, the first operand is really two operands. Execution proceeds as follows: The GPR specified by R1S is fetched, added to the immediate halfword, and the sum is placed into the GPR specified by R1D. Hence it is possible to add a constant to one GPR and put the sum in another. If only one GPR is specified, it is considered to be both R1D and R1S.

For ARH, the operands are a GPR and a main store halfword.

For ARA, the operands are a GPR and the XBD address itself.

For AHR, the operands are a main store halfword and a GPR.

For AHI, the operands are a main store halfword and an immediate halfword.

For AHA, the operands are a main store halfword and the XBD address itself.

For AHH, the operands are two main store halfwords. An alignment procedure check will occur on any instruction specifying a main store halfword not on an even boundary.

8.6 SUBTRACT

Subtract Instructions

The second operand is subtracted from the first, and the difference is placed in the first operand location. The CFR is set as for the add instructions.

The instruction formats are identical to those for the add instructions, except that the operation code is X'*2' rather than X'*1'. Mnemonics are identical except for the first letter, which is "S" rather than "A".

An alignment procedure check will occur on any instruction specifying a main store halfword not on an even boundary.

8.7 MULTIPLY

Multiply Register times Register
MRR   R1,R2

```
┌──────────┬─────┬─────┐
│   03     │ R1  │ R2  │
└──────────┴─────┴─────┘
0          8     12  15
```

Multiply Register times Immediate
MRI   R1D,R1S,IH2 -or- R1,IH2

```
┌──────────┬─────┬─────┬──────────────────┐
│   53     │R1D  │R1S  │       IH2        │
└──────────┴─────┴─────┴──────────────────┘
0          8     12    16                 31
```

Multiply Register times Halfword
MRH   R1,D2(X2,B2)

```
┌──────────┬─────┬─────┬─────┬────────────┐
│   63     │ R1  │ X2  │ B2  │     D2     │
└──────────┴─────┴─────┴─────┴────────────┘
0          8     12    16    20           31
```

Multiply Register times Address
MRA   R1,D2(X2,B2)

```
┌──────────┬─────┬─────┬─────┬────────────┐
│   73     │ R1  │ X2  │ B2  │     D2     │
└──────────┴─────┴─────┴─────┴────────────┘
0          8     12    16    20           31
```

The first operand, which is always a GPR, is multiplied by the second operand to produce a 32-bit product. The low-order sixteen bits of this product are placed into the first operand GPR, and the high-order sixteen bits are placed into the previous GPR (i.e., R1-1). A register specification procedure check occurs if the first operand is GPR0.

For MRR, the second operand is the GPR specified by R2.

MRI is special in that it allows the first operand GPR to be two operands. Execution procedes as follows: the contents of the GPR specified by R1S is multiplied by the immediate halfword. The low-order sixteen bits of the product are placed in the R1D GPR, and the high-order sixteen bits in the previous one. If only one GPR is specified, it is assumed to be both R1S and R1D.

For MRH, the second operand is the specified main store halfword. An alignment procedure check occurs if its address is odd.

For MRA, the second operand is the XBD address itself.

If multiply instructions are used with fractions, the product must be shifted one bit tc the left to give the correct answer.

8.8 DIVIDE

Divide Register by Register
DRR   R1,R2

```
┌──────────┬─────┬─────┐
│    04    │ R1  │ R2  │
└──────────┴─────┴─────┘
0          8     12  15
```

Divide Register by Immediate
DRI   R1D,R1S,IH2 -or- R1,IH2

```
┌──────────┬─────┬─────┬──────────────────┐
│    54    │R1D  │R1S  │       IH2        │
└──────────┴─────┴─────┴──────────────────┘
0          8     12    16                31
```

Divide Register by Halfword
DRH   R1,D2(X2,B2)

```
┌──────────┬─────┬─────┬─────┬────────────┐
│    64    │ R1  │ X2  │ B2  │     D2     │
└──────────┴─────┴─────┴─────┴────────────┘
0          8     12    16    20          31
```

Divide Register by Address
DRA   R1,D2(X2,B2)

```
┌──────────┬─────┬─────┬─────┬────────────┐
│    74    │ R1  │ X2  │ B2  │     D2     │
└──────────┴─────┴─────┴─────┴────────────┘
0          8     12    16    20          31
```

The first operand is the 32-bit dividend, the low-order sixteen bits of which are in the GPR specifed by R1, while the high-order sixteen bits are in the previous GPR. The second operand is the divisor, which is divided into the dividend, producing a quotient, which is placed in the R1 GPR, and a remainder, which is placed in the previous GPR. A register specification procedure check occurs if R1 specifies GPR0.

For DRR, the divisor is in the GPR specified by R2.

The execution of DRI is somewhat different than that of the other divide instructions. It procedes as follows: the dividend is taken from the GPR specified by R1S and the previous one. This dividend is divided by the immediate halfword, and the quotient is placed in the GPR specified by R1D. The remainder is placed in the previous GPR (i.e., R1D-1). If only one GPR is specified, it is assumed to be both R1D and R1S.

For DRH, the divisor is the main store halfword. An alignment procedure check will occur if its address is odd.

For DRA, the divisor is the XBD address itself.

Fractional divides cannot be performed with these instructions.

A division by zero procedure check occurs if the divisor is zero. The CFR is set to indicate overflow: bit 1 is set to zero if there was no overflow, or to one if there was; the summary flag is always zero. Overflow occurs if the quotient is too big to fit in one GPR.

8.9 EXTEND SIGN

(Shift for Divide on M4A)

EXtend Sign of Register
EXSR R

```
┌──────────────┬─────┐
|     0F9      |  R  |
└──────────────┴─────┘
 0             12  15
```

The contents of the specified GPR is treated as a signed integer and extended to 32 bits by replicating its sign in the <u>previous</u> GPR.

A register specification procedure check occurs if GPR0 is specified.

8.10 SQUARE ROOT

SQuare RooT Register
SQRTR R

```
┌──────────────┬─────┐
|     0FA      |  R  |
└──────────────┴─────┘
 0             12  15
```

SQuare RooT Halfword
SQRTH D(X,B)

9FA	X	B	D
0	12	16	20 31

The operand is treated as a signed fraction and replaced by its square root. A negative square root procedure check occurs if the operand is negative.
The operand for SQRTR is the specified GPR, while for SQRTH it is the main store halfword. An alignment procedure check occurs if this halfword is on an odd boundary.

9 COMPARISON INSTRUCTIONS

9.1 ARITHMETIC COMPARES

Compare Register with Register
CRR   R1,R2

```
┌────────┬────┬────┐
│   05   │ R1 │ R2 │
└────────┴────┴────┘
 0        8    12 15
```

Compare Register with Immediate
CRI   R1D,R1S,IH2 -or- R1,IH2

```
┌────────┬────┬────┬────────────────┐
│   55   │R1D │R1S │      IH2       │
└────────┴────┴────┴────────────────┘
 0        8    12   16              31
```

Compare Register with Halfword
CRH   R1,D2(X2,B2)

```
┌────────┬────┬────┬────┬────────────┐
│   65   │ R1 │ X2 │ B2 │     D2     │
└────────┴────┴────┴────┴────────────┘
 0        8    12   16   20          31
```

Compare Register with Address
CRA   R1,D2(X2,B2)

```
┌────────┬────┬────┬────┬────────────┐
│   75   │ R1 │ X2 │ B2 │     D2     │
└────────┴────┴────┴────┴────────────┘
 0        8    12   16   20          31
```

Compare Halfword with Register
CHR   D1(X1,B1),R2

```
┌────────┬────┬────┬────┬────────────┐
│   95   │ R2 │ X1 │ B1 │     D1     │
└────────┴────┴────┴────┴────────────┘
 0        8    12   16   20          31
```

Compare Halfword with Immediate
CHI   D1(X1,B1),IH2

```
┌───────────┬─────┬─────┬─────┬────────────────┬──────────────────────┐
│    B5     │/////│ X1  │ B1  │       D1       │         IH2          │
└───────────┴─────┴─────┴─────┴────────────────┴──────────────────────┘
0           8     12    16    20               32                    47
```

Compare Halfword with Address
CHA   D1(X1,B1),D2(X2,B2)

```
┌───────────┬─────┬─────┬─────┬────────────────┬─────┬────────────────┐
│    C5     │ X2  │ X1  │ B1  │       D1       │ B2  │       D2       │
└───────────┴─────┴─────┴─────┴────────────────┴─────┴────────────────┘
0           8     12    16    20               32    36               47
```

Ccmpare Halfword with Halfword
CHH   D1(X1,B1),D2(X2,B2)

```
┌───────────┬─────┬─────┬─────┬────────────────┬─────┬────────────────┐
│    D5     │ X2  │ X1  │ B1  │       D1       │ B2  │       D2       │
└───────────┴─────┴─────┴─────┴────────────────┴─────┴────────────────┘
0           8     12    16    20               32    36               47
```

The two operands are treated as signed numbers and compared (so that negative numbers are less than zero, which is less than positive numbers). The CFR is set as follows: bit 0 is set if the operands are equal; bit 1 is set if the first operand is greater than the second; or bit 2 is set if it is less. The summary flag is set on if they are equal, off otherwise.

The following extended mnemonics are provided for branching after compares: Branch Equal (BE), Branch Not Equal (BNE), Branch Greater (BG), Branch Not Greater (BNG), Branch Less (BL), and Branch Not Less (BNL).

Note that BH is not an extended mnemonic. BH is a mnemonic for Branch Halfword.

The instructions allow the comparison of all possible combinations of 16-bit numbers in registers, immediate halfwords, addresses, or main store halfwords.

An alignment procedure check occurs if a halfword is specified on an odd bounaary.

9.2 LOGICAL COMPARES

Compare Logical Register with Register
CLRR R1,R2

```
┌──────────┬──────┬──────┐
│    0B    │  R1  │  R2  │
└──────────┴──────┴──────┘
0          8      12    15
```

RR HR BR CC
RI HI BI
RH HH
RA HA
RB

Compare Logical Register with Immediate
CLRI R1,IH2

```
┌──────────┬──────┬──────┬────────────────────┐
│    5B    │  R1  │//////│        IH2         │
└──────────┴──────┴──────┴────────────────────┘
0          8      12     16                  31
```

Compare Logical Register with Halfword
CLRH R1,D2(X2,B2)

```
┌──────────┬──────┬──────┬──────┬──────────────┐
│    6B    │  R1  │  X2  │  B2  │      D2      │
└──────────┴──────┴──────┴──────┴──────────────┘
0          8      12     16     20            31
```

Compare Logical Register with Address
CLRA R1,D2(X2,B2)

```
┌──────────┬──────┬──────┬──────┬──────────────┐
│    7B    │  R1  │  X2  │  B2  │      D2      │
└──────────┴──────┴──────┴──────┴──────────────┘
0          8      12     16     20            31
```

Compare Logical Register with Byte
CLRB R1,D2(X2,B2)

```
┌──────────┬──────┬──────┬──────┬──────────────┐
│    8B    │  R1  │  X2  │  B2  │      D2      │
└──────────┴──────┴──────┴──────┴──────────────┘
0          8      12     16     20            31
```

Compare Logical Halfword with Register
CLHR D1(X1,B1),R2

```
┌──────────┬──────┬──────┬──────┬──────────────┐
│    9B    │  R2  │  X1  │  B1  │      D1      │
└──────────┴──────┴──────┴──────┴──────────────┘
0          8      12     16     20            31
```

Compare Logical Byte with Register
CLBR D1(X1,B1),R2

```
┌──────────┬──────┬──────┬──────┬─────────────────┐
│    AB    │  R2  │  X1  │  B1  │       D1        │
└──────────┴──────┴──────┴──────┴─────────────────┘
0          8      12     16     20               31
```

Compare Logical Halfword with Immediate
CLHI D1(X1,B1),IH2

```
┌──────────┬──────┬──────┬──────┬───────────────┬───────────────────┐
│    BB    │//////│  X1  │  B1  │      D1       │        IH2        │
└──────────┴──────┴──────┴──────┴───────────────┴───────────────────┘
0          8      12     16     20              32                 47
```

Compare Logical Halfword with Address
CLHA D1(X1,B1),D2(X2,B2)

```
┌──────────┬──────┬──────┬──────┬───────────────┬──────┬────────────┐
│    CB    │  X2  │  X1  │  B1  │      D1       │  B2  │     D2     │
└──────────┴──────┴──────┴──────┴───────────────┴──────┴────────────┘
0          8      12     16     20              32     36          47
```

Compare Logical Halfword with Halfword
CLHH D1(X1,B1),D2(X2,B2)

```
┌──────────┬──────┬──────┬──────┬───────────────┬──────┬────────────┐
│    DB    │  X2  │  X1  │  B1  │      D1       │  B2  │     D2     │
└──────────┴──────┴──────┴──────┴───────────────┴──────┴────────────┘
0          8      12     16     20              32     36          47
```

Compare Logical Byte with Immediate
CLBI D1(X1,B1),IB2

```
┌──────────┬──────┬──────┬──────┬───────────────┬──────────┬────────┐
│    EB    │//////│  X1  │  B1  │      D1       │//////////│  IB2   │
└──────────┴──────┴──────┴──────┴───────────────┴──────────┴────────┘
0          8      12     16     20              32         40      47
```

Compare Logical Character string with Character string
CLCC D1(X1,B1),D2(X2,B2),DL(BL)

```
┌──────────┬──────┬──────┬──────┬─────────┬──────┬─────────┬──────┬──────┐
│    FB    │X2    │X1    │B1    │ D1      │B2    │ D2      │BL    │  DL  │
└──────────┴──────┴──────┴──────┴─────────┴──────┴─────────┴──────┴──────┘
0          8      12     16     20        32     36        48     52    63
```

The two operands are treated as logical bit strings and compared in magnitude (so that zero is the smallest number). The CFR is set as for the other compare instructions.

These instructions allow all possible combinations of byte-byte, halfword-halfword, and character string-character string comparisons.

An alignment procedure check occurs if a halfword is specified on an odd boundary.

10 BIT AND CHARACTER STRING INSTRUCTIONS

RR HR BR CC
RI HI BI
RH HH
RA HA
RB

10.1 OR

Or Register with Register
ORR R1,R2

```
┌──────────┬──────┬──────┐
|   08     | R1   | R2   |
└──────────┴──────┴──────┘
0          8      12   15
```

Or Register with Immediate
ORI R1D,R1S,IH2 -or- R1,IH2

```
┌──────────┬──────┬──────┬──────────────────────┐
|   58     |R1D   |R1S   |         IH2          |
└──────────┴──────┴──────┴──────────────────────┘
0          8      12     16                     31
```

Or Register with Halfword
ORH R1,D2(X2,B2)

```
┌──────────┬──────┬──────┬──────┬──────────────────┐
|   68     | R1   | X2   | B2   |       D2         |
└──────────┴──────┴──────┴──────┴──────────────────┘
0          8      12     16     20                 31
```

Or Register with Address
ORA R1,D2(X2,B2)

```
┌──────────┬──────┬──────┬──────┬──────────────────┐
|   78     | R1   | X2   | B2   |       D2         |
└──────────┴──────┴──────┴──────┴──────────────────┘
0          8      12     16     20                 31
```

Or Register with Byte
ORB R1,D2(X2,B2)

```
┌──────────┬──────┬──────┬──────┬──────────────────┐
|   88     | R1   | X2   | B2   |       D2         |
└──────────┴──────┴──────┴──────┴──────────────────┘
0          8      12     16     20                 31
```

Or Halfword with Register
OHR D1(X1,B1),R2

```
|    98    | R2 | X1 | B1 |      D1      |
0          8    12   16   20            31
```

Or Byte with Register
OBR D1(X1,B1),R2

```
|    A8    | R2 | X1 | B1 |      D1      |
0          8    12   16   20            31
```

Or Halfword with Immediate
OHI D1(X1,B1),IH2

```
|    B8    |////| X1 | B1 |      D1      |      IH2      |
0          8    12   16   20            32              47
```

Or Halfword with Address
OHA D1(X1,B1),D2(X2,B2)

```
|    C8    | X2 | X1 | B1 |      D1      | B2 |    D2    |
0          8    12   16   20            32   36         47
```

Or Halfword with Halfword
OHH D1(X1,B1),D2(X2,B2)

```
|    D8    | X2 | X1 | B1 |      D1      | B2 |    D2    |
0          8    12   16   20            32   36         47
```

Or Byte with Immediate
OBI D1(X1,B1),IB2

```
|    E8    |////| X1 | B1 |      D1      |/////////| IB2 |
0          8    12   16   20            32        40    47
```

Or Character string with Character string
OCC D1(X1,B1),D2(X2,B2),DL(BL)

```
|    F8    |X2  |X1  |B1  | D1     |B2  | D2     |BL  | DL     |
0          8    12   16   20       32   36       48   52      63
```

The two operands are treated as logical bit strings, and a boolean 'OR' is performed upon them. The result is placed in the first operand location.

The instructions allow all possible combinations of byte-byte, halfword-halfword, and character string-character string operations.

An alignment procedure check occurs if a halfword is specified on an odd boundary.

10.2 AND INSTRUCTIONS

The two operands are treated as logical bit strings, and a boolean 'AND' is performed upon them. The result is placed in the first operand location.

The instruction formats are identical to those for the OR instructions, except that the operation code is X'*9' rather than X'*8'. Mnemonics are identical except for the first letter, which is "N" rather than "O".

An alignment procedure check occurs if a halfword is specified on an odd boundary.

10.3 EXCLUSIVE OR INSTRUCTION

The two operands are treated as logical bit strings, and a boolean 'EXCLUSIVE OR' is performed upon them. The result is placed in the first operand location.

The instruction formats are identical to those for the OR instructions, except that the operation code is X'*A' rather than X'*8'. Mnemonics are identical except for the first letter, which is "X" rather than "O".

An alignment procedure check occurs if a halfword is specified on an odd boundary.

10.4 TEST UNDER MASK INSTRUCTIONS

The second operand is used as a mask to select bits in the first operand. Each one bit in the mask selects the corresponding bit in the first operand. The CFR is set as follows: bit 0 is set if all selected bits are zero or the mask is all zeroes; bit 1 is set if all selected bits are one; or bit 2 is set if the selected bits are mixed zeroes and ones. The summary flag is set on if all selected bits are one; it is set off otherwise.

Extended mnemonics are provided for branching after a Test under Mask instruction: Branch all Zeroes (BZ), Branch Not all Zeroes (BNZ), Branch all Ones (BO), Branch Not all Ones (BNO), Branch Mixed (BM), and Branch not Mixed (BNM).

The instruction formats are identical to those for the boolean instructions, except that the operation code is X'*C' rather than X'*8,9,A'. Mnemonics are identical except for the first letter which is "TM" rather than "O,N,X". Additionally, the TMCC instruction (for testing character strings) is not provided.

An alignment procedure check occurs if a halfword is specified on an odd boundary.

10.5 SCAN

SCan Forward Equal to Addres
SCFEA D1(X1,B1),D2(X2,B2),DL(BL)

```
|    F0    |X2   |X1   |B1   | D1       |B2   | D2        |BL   |   DL   |
0          8     12    16    20         32    36          48    52      63
```

SCan Backward Equal to Address
SCBEA D1(X1,B1),D2(X2,B2),DL(BL)

```
|    F2    |X2   |X1   |B1   | D1       |B2   | D2        |BL   |   DL   |
0          8     12    16    20         32    36          48    52      63
```

The first operand character string is scanned forward (left to right) or backward (right to left) for a character equal to that specified by the low-order byte of the second operand address. If the character is found, bit 0 of the CFR is set on and GPR1 is set to point at that character in operand 1. If no such character is present, bit 1 of the CFR is set on and GPR1 is unchanged. The summary flag is set on if successful, off otherwise.

Extended mnemonics are provided for branching after a scan: Branch Successful (BS), and Branch Not Successful (BNS).

If the length is zero, the instruction always fails.

SCan Forward Not equal to Address
SCFNA D1(X1,B1),D2(X2,B2),DL(BL)

```
┌──────────┬─────┬─────┬─────┬──────────┬─────┬──────────┬─────┬──────────┐
│    F1    │X2   │X1   │B1   │ D1       │B2   │ D2       │BL   │    DL    │
└──────────┴─────┴─────┴─────┴──────────┴─────┴──────────┴─────┴──────────┘
 0          8     12    16    20         32    36         48    52       63
```

SCan Backward Not equal to Address
SCBNA D1(X1,B1),D2(X2,B2),DL(BL)

```
┌──────────┬─────┬─────┬─────┬──────────┬─────┬──────────┬─────┬──────────┐
│    F3    │X2   │X1   │B1   │ D1       │B2   │ D2       │BL   │    DL    │
└──────────┴─────┴─────┴─────┴──────────┴─────┴──────────┴─────┴──────────┘
 0          8     12    16    20         32    36         48    52       63
```

The first operand character string is scanned forward (left to right) or backward (right to left) for a character not equal to that specified by the low-order byte of the second operand address. If an unequal character is found, bit 0 of the CFR is set on and GPR1 is set to point at that character in operand 1. If all characters are equal, bit 1 of the CFR is set on and GPR1 is unchanged. The summary flag is set on if successful, off otherwise.

Extended mnemonics are provided for branching after a scan: Branch Successful (BS), and Branch Not Successful (BNS).

If the length is zero, the instruction always fails.

SCan Forward using Table
SCFT D1(X1,B1),D2(X2,B2),DL(BL)

```
┌──────────┬─────┬─────┬─────┬──────────┬─────┬──────────┬─────┬──────────┐
│    F4    │X2   │X1   │B1   │ D1       │B2   │ D2       │BL   │    DL    │
└──────────┴─────┴─────┴─────┴──────────┴─────┴──────────┴─────┴──────────┘
 0          8     12    16    20         32    36         48    52       63
```

SCan Backward using Table
SCBT D1(X1,B1),D2(X2,B2),DL(BL)

```
┌──────────┬─────┬─────┬─────┬──────────┬─────┬──────────┬─────┬──────────┐
│    F5    │X2   │X1   │B1   │ D1       │B2   │ D2       │BL   │    DL    │
└──────────┴─────┴─────┴─────┴──────────┴─────┴──────────┴─────┴──────────┘
 0          8     12    16    20         32    36         48    52       63
```

The first operand character string is scanned forward or backward a character at a time. Each character is used to select a byte from the second operand table. If this selected byte is zero, scanning continues. If this byte is non-zero, it is placed in GPR0 and the address of the selector character in operand 1 is placed in GPR1. If the scan is successful, CFR bit 0 iand the summary flag are set on; otherwise CFR bit 1 is set on and the summary flag is set off (allowing use of the BS and BNS extended mnemonics).

The second operand table is always 256 bytes long. Hence there is one table byte for every possible value of a character from operand 1. Table bytes are selected by taking the "n"th table byte if the value of the operand 1 character is "n".

If the length is zerc, the instruction always fails.

10.6 TRANSLATE

TRanslate character string
TR   D1(X1,B1),D2(X2,B2),DL(BL)

FD	X2	X1	B1	D1	B2	D2	BL	DL
0	8	12	16	20	32	36	48	52 63

The first operand character string is translated according to the 256-byte table specified by operand 2. Each character in operand 1 is scanned, its value , say "n", is used to select the correspoding "n"th byte in the table, which then replaces the old operand 1 byte.
If the length is zerc, no translation is performed.

10.7 INITIALIZE

INITialize character string
INIT   D1(X1,B1),D2(X2,B2),DL(BL)

FE	X2	X1	B1	D1	B2	D2	BL	DL
0	8	12	16	20	32	36	48	52 63

The first operand character string is initialized to the low-order byte of the operand 2 address. The whole character string is filled with this byte.
If the length is zero, no initialization is performed.

11 SUBROUTINES AND INTERRUPTS

Procedures were first mentioned in Chapter 3.1 -- how to write them and how they were controlled. This chapter gives a more detailed explanation, particularly in terms of how procedure *calls* are performed.

A procedure call can occur for two reasons. First the programmer may request an explicit *subroutine call* via the CALL instruction. Secondly, the q-interpreter might force an implicit *interrupt call* when an interrupt request is detected. Both of these calls are performed in the same manner, so that a called procedure need not know whether it is a subroutine or an interrupt handler.

11.1 THE STACK

In order to maintain the correct sequence of procedure calls, and to save the contents of various registers across these calls, a save area *stack* is employed. It has the following format:

```
                    .
                    .
                    .
             ┌─────────────────────┐
             |      SAVE AREA      |
          42 |          1          |
             ├─────────────────────┤
             |      SAVE AREA      |
           2 |          0          |
             ├──────────┬──────────┤
   SP-->   0 | current  |   last   |
             └──────────┴──────────┘
             0          8         15
```

As a new procedure is invoked, it is assigned the next available save area on the stack. The save area number for the currently executing procedure is contained in the "current" byte. A save area is twenty halfwords long, and is used to save the general-purpose registers, PBR, PDR, CFR, and ICMR (see below) whenever a new procedure is called. When the procedure returns, they are restored, thus preserving their contents across the call.

The "last" byte gives the save area number of the last useable save area on the stack. This helps prevent main store following the stack from being used indiscriminately.

META 4B control register 20, the Stack Pointer (SP), is always set to point at the stack ("current" byte). This register should not be tampered with by the user.

11.2 STACK CONDITIONS

Certain special conditions concerning the save area stack can arise during program execution. These conditions require special consideration (probably performed by a monitor program) and are described here.

11.2.1 STACK OVERFLOW

When the q-interpreter determines that a procedure being invoked will own the next-to-last save area, it considers stack overflow to have occurred. After invocation is complete, but before the first instruction of the new procedure is executed, a stack overflow procedure check will be forced. This allows a monitor program to allocate a bigger stack.

11.2.2 STACK ESCAPE

If the warning given by stack overflow goes unheeded, a procedure may eventually be invoked which can own no save area. If this procedure were to call another, or an interrupt call were to occur, stack escape would be recognized. In this case, the q-interpreter goes into an infinite loop.

11.2.3 STACK UNDERFLOW

If an attempt is made to return from a procedure which owns the 0th save area, stack underflow occurs. A stack underflow procedure check is generated, and a monitor program can react as desired.

11.3 PROCEDURE CALLS

CALL via Register
CALLR R

```
┌──────────────┬─────┐
│     0FD      │  R  │
└──────────────┴─────┘
 0             12  15
```

CALL via Address
CALLA D(X,B)

```
┌──────────────┬─────┬─────┬───────────────┐
│     7FD      │  X  │  B  │       D       │
└──────────────┴─────┴─────┴───────────────┘
 0             12    16    20             31
```

CALL via Halfword
CALLH D(X,B)

```
┌──────────────┬─────┬─────┬───────────────┐
│     9FD      │  X  │  B  │       D       │
└──────────────┴─────┴─────┴───────────────┘
 0             12    16    20             31
```

The operand specifies the address of a procedure to be invoked. The following steps are taken by the q-interpreter:

1. Check for stack escape condition.

2. Save general-purpose registers, PBR, PDR, CFR, and ICMR in current save area.

3. Assign next save area to new procedure, and update "current" byte.

4. Update PBR and GPR15 to point at new procedure, and set PDR to zero.

5. Cause stack overflow procedure check if appropriate, or begin execution of new procedure.

An alignment procedure check cccurs if the procedure address is odd.

11.4 INTERRUPT CALLS

An interrupt call is performed automatically by the q-interpreter when some special event occurs of which the software must be informed. The interrupt procedure is invoked almost exactly like a called procedure, except that its address cannot be provided by the programmer at the time of invocation. Instead, these addresses are obtained from the stack prefix, which resides below the stack in main store. The stack prefix contains pairs of halfwords, one for each interruption source. It has the following format:

```
SP-->|                    |
     |--------------------|
  -4 | Extended Inst.     |
     |--------------------|
  -8 | Self-interrupt     |
     |--------------------|
 -12 | Procedure Check    |
     |--------------------|
 -16 |     META 4A        |
     |--------------------|
 -20 |     SIMALE         |
     |--------------------|
 -24 | Vector General     |
     |____________________|
```

The first halfword of each pair contains the address of a procedure to handle interrupts from that interrupt source. The second halfword contains the ICMR which should be in affect during execution of the interrupt procedure.

The Interrupt Call Mask Register (ICMR), control register 19, contains bits which control the occurence of interrupt calls. It has the following format:

```
ICMR

 ________________________________
|VG bits |A|/////|QQT|
|________|_|_____|___|
0         8      13 15
```

Bits 0-8 are used to control interrupt calls for the META 4A and Vector General. Whenever these units request an interrupt, one of these bits is checked. If it is off, the interrupt is ignored for the time being. If it is on, an interrupt call is performed. More detail will be presented in later chapters.

Bits 13 and 14 are used by the q-interpreter and should never be set on by the user.

Bit 15 is the trace bit. See the META 4B q-interpreter listing for a description of this bit.

The following steps are performed by the q-interpreter when an interrupt procedure is to be called:

1. Check for stack escape condition.

2. Save general-purpose registers, PBR, PDR, CFR, and ICMR in the current save area.

3. Assign next save area to new procedure, and update "current" byte.

4. Update PBR, GPR15, and ICMR from stack base, and set PDR to zero.

5. Put interrupt designator in GPR6, and any interrupt information in GPR7 on up.

6. Cause stack overflow procedure check if appropriate, or begin execution of new procedure.

The interrupt designator in Step 5 is a halfword specifying the source of the interruption. This and the interrupt information will be described in detail in the appropriate chapters.

11.5 PROCEDURE RETURN

RETURN from procedure
RETURN   first,count

```
┌─────────┬─────┬─────┬──────────────────────┐
|    5E   |frst |frst |        count         |
└─────────┴─────┴─────┴──────────────────────┘
0         8     12    16                    31
```

The RETURN instruction is used to terminate the execution of a procedure and return to the previously-executing one (caller or interrupted). A set of general-purpose registers may be preserved across this return, so that results or completion codes may be returned to the caller. The first register and number of registers to be preserved are specified by the "first" and "count" operands, respectively. If the count is zero, no registers are preserved.
The following steps are taken by the q-interpreter:

1. Check for stack underflow, and cause a procedure check if it exists.

2. Update "current" byte to specify previous save area.

3. Restore general-purpose registers (except those to be preserved), and PBR, PDR, CFR, and ICMR.

4. Continue execution in the previous procedure.

12 INTERNAL INTERRUPTS

Interrupt requests can be divided up into two classes: internal and external. Internal interrupts originate from within the q-interpreter, and are necessary to indicate various exceptional programming requirements or errors. External interrupts are generated by units other than the META 4B, and are used to inform the B of special conditions in those units.
This chapter deals with the three kinds of internal interrupts: extended instruction interrupts, self-interrupts, and procedure checks.

12.1 EXTENDED INSTRUCTION INTERRUPTS

Extended instructions are an optional facility useful for interfacing between a user's program and the monitor. They allow the user to code monitor requests in the normal instruction format, using otherwise invalid operation codes, and to have these "invalid" instructions trapped and interpreted by software. This technique is extensively used in the META 4A.

In order for an operation code to be considered extended rather than invalid, alterations must be made to a table in the q-interpreter. Given these alterations, the occurence of an extended operation code causes an immediate extended instruction interrupt call. The interrupt procedure will find the following information in its general-purpose registers:

GPR6: X'0000'
GPR7: first halfword of instruction.
GPR8: contents of q-interpreter register A1.
GPR9: contents of q-interpreter register D1.
GPR10: contents of q-interpreter register A2.
GPR11: contents of q-interpreter register D2.

The contents of the four q-interpreter registers depends upon the format of the extended instruction. See the q-interpreter listing for more detail.

12.2 SELF-INTERRUPTS

Self-INTerrupt via Register
SINTR   R

```
┌──────────────────┬──────┐
|      0FE         |  R   |
└──────────────────┴──────┘
0                  12  15
```

Self-INTerrupt via Halfword
SINTH   D(X,B)

```
┌──────────────────┬──────┬──────┬─────────────────┐
|      9FE         |  X   |  B   |        D        |
└──────────────────┴──────┴──────┴─────────────────┘
0                  12     16     20               31
```

Self-INTerrupt via Address
SINTA   D(X,B)

```
┌──────────────────┬──────┬──────┬─────────────────┐
|      7FE         |  X   |  B   |        D        |
└──────────────────┴──────┴──────┴─────────────────┘
0                  12     16     20               31
```

The self-interrupt instructions provide another, more explicit, means of communicating with the monitor. Upon execution of one of these instructions, an immediate self-interrupt call is performed, with the interrupt procedure receiving the following information:

GPR6: X'0001'
GPR7: instruction operand.

12.3 PROCEDURE CHECKS

A cursory description cf procedure checks was given in Chapter 3.3. They provide a means for informing the user and/or monitor of programming errors. With each instruction description in this document, the possible procedure checks are noted.

When a procedure check occurs, the PDR is backed up to point at the instruction in error, and a procedure check interrupt call is forced. The interrupt procedure receives the following information:

GPR6: X'0002'
GPR7: procedure check type code:
0 - invalid operation code

1 - alignment
2 - register specification
3 - division by zero
4 - SQRT operand negative
5 - stack overflow
6 - stack underflow

13 COMMUNICATION WITH THE META 4A

13.1 OVERVIEW

Communication with the META 4A is a necessary ingredient for performing multiprocessing tasks on BUGS. Facilities are provided for interrupting each processor from the other, and for synchronizing execution via Dijkstra-like semaphores.

13.2 THE UNIT CONTROL BLOCKS

Interrupting of one processor by the other is controlled by the Unit Control Blocks (UCB). The META 4A is unit 7; hence its UCB pointer is at locaticn X'3E' (since the UCB pointer table starts at locations X'30'). The META 4B is unit 1; its UCB pointer is at location X'32'. The formats of the two UCBs are as follows:

META 4A UCB

```
  ┌──────────────────┐
0 │                 7│
  ├──────────────────┤
2 │       USH        │
  └──────────────────┘
   0               15
```

META 4B UCB

```
  ┌──────────────────┐
0 │                 1│
  ├──────────────────┤
2 │       USH        │
  ├──────────────────┤
4 │  start-up PBR    │
  ├──────────────────┤
6 │  start-up ICMR   │
  ├──────────────────┤
8 │  start-up SP     │
  └──────────────────┘
   0               15
```

13.3 META 4A INTERRUPTS META 4B

The META 4A interrupts the B via the extended instruction INTB:

```
INTB        D(X,B)

┌──────────┬─────┬─────┬─────┬─────────────┐
│    6F    │/////│  X  │  B  │      D      │
└──────────┴─────┴─────┴─────┴─────────────┘
 0          8     12    16    20          31
```

The operand address is used as the interrupt code. There are two classes of interrupts:

13.3.1 Q-INTERPRETER-HANDLED INTERRUPTS

If bit 0 of the interrupt code is on, the META 4B's q-interpreter handles the interrupt and does not give it to the software. The only q-interpreter interrupt present now is code X'8000'. This is a start-up interrupt and causes the B to perform the following steps:

1. Halt execution of any current program.

2. Pick up the start-up PBR from its UCB, copy it into GPR15, and zero the PDR.

3. Zero the CFR.

4. Pick up the start-up ICMR and SP from its UCB.

5. Begin execution of the new procedure. This is the method by which the META 4A starts up a program in the META4B.

13.3.2 SOFTWARE-HANDLED INTERRUPTS

If bit 0 of the interrupt code is off, the q-interpreter will attempt to cause a META 4A interrupt call. Bit 8 in the current ICMR controls this call -- if it is on, then an interrupt call immediately occurs; if it is off, the interrupt remains pending. When the interrupt procedure receives control, the resgiters are loaded with the following information:

GPR6: X'8007'
GPR7: the interrupt code

<u>NOTE</u> <u>CAREFULLY</u>: Software interrupt codes are divided up into two classes for purposes of the monitor environment. Users should restrict their interrupt codes to the range 0 to X'3FFF'. All codes above X'3FFF' are trapped by the monitor and not given to the user.

13.4 META 4B INTERRUPT META 4A

The META 4B interrupts the A via the instructions:

INTerrupt A via Register
INTAR R

```
┌───────────────┬─────┐
│      0FF      │  R  │
└───────────────┴─────┘
 0              12  15
```

INTerrupt A via Halfword
INTAH D(X,B)

```
┌───────────────┬─────┬─────┬─────────────────┐
│      9FF      │  X  │  B  │        D        │
└───────────────┴─────┴─────┴─────────────────┘
 0              12    16    20               31
```

INTerrupt A via Address
INTAA D(X,B)

```
┌───────────────┬─────┬─────┬─────────────────┐
│      7FF      │  X  │  B  │        D        │
└───────────────┴─────┴─────┴─────────────────┘
 0              12    16    20               31
```

The instruction operand is used as the interrupt code to the A. As with the INTB instruction, users should restrict themselves to codes between 0 and X'3FFF'.

The methods by which the user actually specifies interrupt procedures on the A and B is described in Chapter 17.

13.5 SYNCHRONIZATION VIA SEMAPHORES

Semaphores are a means of synchronizing execution of parallel processors. A semaphore on BUGS is a halfword in main store which is usually associated with a resource that both processors wish to use (e.g., a data structure). Bit 15 of this semaphore determines whether or not the resource is free -- if it is off, the resource is free and can be used by a processor; if it is on, the resource is already in use.

A typical sequence of code utilizing a semaphore to control a resource would be as follows:

1. test bit 15 of the semaphore.

2. if on, repeat step 1. if off, set it on and continue.

3. use the resource.

4. reset bit 15 to zero

Steps 1 and 2 must be performed so that the other processor cannot change the semaphore in between them. The B provides a special instruction to do this:

```
SEMAphore
SEMA        D(X,B)
┌────────────┬─────┬─────┬────────────┐
│     9D     │  X  │  B  │     D      │
└────────────┴─────┴─────┴────────────┘
0            12    16    20          31
```

Bit 15 of the halfword operand is tested. If it is on, bit 1 of the CFR is set; if off, bit 0 is set. The summary flag is set to reflect the on state. The bit 15 is then unconditionally set to one. The two steps are done with main store locked out, so that the A cannot tamper with the semaphore.

If the semaphore address is odd, an alignment procedure check occurs.

Thus, to use a resource on the B, the user codes:

```
  SEMA      <semaphore>
  BT        *-4
   •
   • use the resource
   •
  SZH       <semaphore>
```

The same sequence would be coded on the A as:

```
  TSL       <semaphore>+1,X'01'
  BO        *-4
   •
   • use the resource
   •
```

```
     LZ        R2,<semaphore>
```

with the TSL instruction performing the same duties as the SEMA.

14 THE SIMALE

15 VECTOR GENERAL

15.1 (VECTOR) GENERAL DESCRIPTION

15.1.1 INTRODUCTION

The Vector General (VG) is an I/O unit connected to the META 4B and capable of processing numerous types of graphical information. As its main function, it displays graphical data upon a Cathode-Ray Tube (CRT), not unlike an oscilloscope. This data is in the form of point, line and character specifications. Additionally, the VG is equipped with various interactive devices useful for inputting of information by the user.

15.1.2 CRT DISPLAY UNIT

The cathode-ray tube creates images by moving a beam of electrons across a glass face covered with phosphorus. This beam traces out the points and lines making up the image, as specified by <u>orders</u> to the VG control logic. It is necessary to draw the picture continually -- about 40 times per second -- to maintain a steady image without flicker.

Each point on the scope face is represented by three coordinates: X and Y to select the position on the screen, and Z to select the writing intensity. +Z is considered to be in front of the scope face, while -Z is behind it. It is possible for the user to work in two dimensions, maintaining a constant Z intensity, or to work in all three dimensions.

A coordinate is a 12-bit signed number, most commonly treated as a fraction from -1.0 to +.999... It is also possible to consider it an integer from -2048 to +2047, although this can lead to problems. Thus our screen consists of a 4096 X 4096 X 4096 grid of points, called <u>raster units</u>. The cubic space created by these coordinates is called the <u>scope</u> or <u>image space</u>, with the point (0,0,0) located in the center of the scope face.

15.1.2.1 VECTORS

The VG is capable of drawing vectors from the current beam position to any specified point. These vectors can be blanked (that is, the beam can be turned off) or non-blanked, allowing both beam movement and drawing. Furthermore, a drawn line can be solid, dashed, dotted, or simply an end-point.

15.1.2.2 CHARACTERS

Characters can be drawn on the scope in four different sizes. The character set consists of about 200 graphics, represented by 8-bit ASCII codes. Appendix 3a contains a table of the character set.

A character occupies a certain amount of space in the scope space, measured in raster units. The following table describes the four sizes:

CODE	ROWS	COLUMNS	WIDTH	HEIGHT		
0	60	120	34	68	.008	.017
1	40	80	50	100	.012	.024
2	30	60	68	136	.017	.033
3	16	32	128	256	.031	.063
	(per screen)		(raster units)			

Certain ASCII codes perform control functions, such as carriage return or changing the character size. These characters are given symbolic names, which are generated by the M4BEQUS macro if the argument "VG" is specified.

Each ASCII code has an equivalent EBCDIC code and vice-versa (see Appendices 3a and 3b). Translate tables are available for converting these codes, and are generated as follows:

```
     TABLE ASCII-EBCDIC | EBCDIC-ASCII
```

15.1.3 INTERACTIVE DEVICES

The VG is equipped with a veritable plethora of devices which can be employed by the user to input information. These devices are located on the table in front of the CRT display and are manipulated by the human hands. The following paragraphs describe these devices:

Joystick. The joystick is a small spring-loaded upright shaft with three degrees of freedom: left/right, back/forth, and twist. These are most commonly thought of as representing movement in the X, Y, and Z dimensions, respectively. The analog settings of these three degrees of freedom are passed through an A/D converter, and are continuously available to the user as three 12-bit signed fractions.

Dials. There are ten analog dials (numbered 1-10) on a small box connected to the VG. The analog settings of these dials are passed through an A/D converter, and are continuously available to the user as ten 12-bit signed fractions.

Light Pen. The light pen is a small, pen-shaped device which can be held by the user. If the pen is pointed at the CRT screen, and the electron beam passes in front of it, a signal is generated to the VG. If appropriately enabled, this signal can result in an interrupt request to the META 4B.

The light pen is also equipped with a finger-activated tip switch, which can be used in conjunction with the pen to control interrupt requests. This is explained in greater detail below.

Data Tablet. The data tablet is a flat tablet-sized device which can produce X-Y coordinates when tracked over by a stylus. In addition, if appropriately enabled, it produces interrupts to inform the user of the location of the stylus above the table surface.

The data tablet is good for inputting free-form pictures or characters to the program.

Keyboard. A rather complex alphanumeric keyboard is available for inputting characters. Appendix 3c shows the physical layout of this keyboard, with the graphics produced by each key. The keyboard produces ASCII character codes identical to those used by the character drawing facility.

If appropriately enabled, depression of a key causes an interrupt request to the META 4B. If held down, any key will cause a request every sixth of a second.

Function Keys. A panel of 33 function keys (numbered 0-32) is available on the VG. Each key is a small button which has the capability of being illuminated. If appropriately enabled, depression of a function key causes an interrrupt request to the META 4B. The B can then determine which key was hit. Key 32 is special in that it will generate continuous interrupt requests.

Any combination of function keys 0-31 (but not 32) can be illuminated by the user under program control.

15.2 VG REGISTER FILE

As mentioned in Chapter 2.2.4, the VG is equipped with a register file, which can be referenced with the XFER instruction. These registers vary in length from eight to twelve bits, but are always transferred in a 16-bit halfword, with relevent bits left-justified.

The following paragraphs describe the register file. Each register is assigned a symbclic name by which it should be referenced. These names are defined by the M4BEQUS macro if the argument "VG" is given. In addition each register has a 7-bit integer address.

X COORDinate (VGXCOORD, 8)
Y COORDinate (VGYCOORD, 9)
Z COORDinate (VGZCOORD, 10)

```
+-------------+
| fraction    |
+-------------+
 0          11
```

These three registers always contain the 12-bit signed fractional coordinates of the current beam position. They can be fetched and loaded. If loaded, they cause the beam to move (not draw) to the specified position. This is useful for setting up initial image starting points.

X SCALE (VGXSCALE, 17)
Y SCALE (VGYSCALE, 19)

```
+-------------+
| fraction    |
+-------------+
 0          11
```

These two registers are used to scale the X and Y dimensions of the image. The 12-bit signed fractions in these registers are multiplied times any X and Y image coordinates, respectively, before they are used for moving the beam. If a scale register contains a negative number, the image will be scaled and inverted in that dimension.

Notice that the two registers are not contiguously numbered.

Z or Intensity Scale Register (VGISR, 13)

```
┌──────────────┐
| fraction     |
└──────────────┘
 0           11
```

When this register contains zero, the writing intensity is full, and all Z coordinates are ignored, allowing the user to work in only two dimensions.

for gradation
bright + dim,
set ISR = 7FF0
dim = A001
bright = 0

When this register contains a negative fraction, all points with a +Z coordinate are written at full intensity, while those with a -Z coordinate are gradated. The smaller the Z coordinate, the dimmer it is drawn.

When this register contains a positive fraction, all points with a +Z coordinate are blanked, while those with a -Z coordinate are gradated.

The magnitude of the ISR determines the range of gradation. The bigger the ISR is in magnitude, the more gradation that occurs.

X DISPlacement (VGXDISP, 20)
Y DISPlacement (VGYDISP, 21)

```
┌──────────────┐
| fraction     |
└──────────────┘
 0           11
```

ISR = 8000
ISR = C000, (-.5
goes + less ½
dimmer
A001 is good for dim
(+.75)
0, - go bright

These two registers are used for displacing the image to be drawn. After scaling is performed, these 12-bit signed fractions are added to the resulting coordinates before they are used to move the beam. This can be used to display an image centered around a point other than (0,0).

JOYstick X coordinate (VGJOYX, 67)
JOYstick Y coordinate (VGJOYY, 68)
JOYstick Z coordinate (VGJOYZ, 69)

```
┌──────────────┐
| fraction     |
└──────────────┘
 0           11
```

when VR13 = 4000,
8000 is dim,
but grade
F000 dim
[illegible]

These three registers contain the digitized 12-bit signed fractional values of the three joystick degrees-of-freedom. Storing into these registers is ignored.

DIAL 1 value (VGDIAL1, 70)

DIAL 2 value (VGDIAL2, 71)

...

DIAL 10 value (VGDIAL10, 79)

```
┌────────────────┐
| fraction       |
└────────────────┘
 0              11
```

These ten registers contain the digitized 12-bit signed fractional values of each of the ten dials. Storing into these registers is ignored.

data TABlet X coordinate (VGTABX, 2)

```
┌────────────────┬───┬─┬─┐
|  fraction      |///|I|T|
└────────────────┴───┴─┴─┘
 0              11   14 15
```

data TABlet Y coordinate (VGTABY, 3)

```
┌────────────────┐
| fraction       |
└────────────────┘
 0              11
```

The high-order 12 bits of these two registers contain the digitized signed fractional values of the X and Y sensing circuits in the data tablet. In addition, the VGTABX register contains two position bits:

I: This bit is on when the stylus is less than one inch &Sabove the tablet surface; it is off otherwise.

T: This bit is on when the stylus is touching the tablet surface; it is off otherwise.

If appropriately enabled, an interrupt request is made to the B whenever either of these bits changes.

Storing into these registers is ignored.

Mode Control Register (VGMCR, 5)

```
┌─┬─┬─┬─┬─┬─┬─┬─┬────┐
|1|1|L|D|K|F|B|0|1111|
└─┴─┴─┴─┴─┴─┴─┴─┴────┘
 0 1 2 3 4 5 6 7 8  11
```

The L, D, K, and F bits control the light pen, data tablet, keyboard, and function keys, respectively. If a bit is off, the corresponding device is entirely disabled. If it is on, use of the device will cause interrupt requests to the META 4B.

The blink bit (B), if on, causes any image being displayed to blink nauseatingly every half second (bad karma).

The remaining bits must be set as specified in the picture.

Function Key Lights 0-7 (VGFKL0, 0)
Function Key Lights 8-15 (VGFKL8, 1)
Function Key Lights 16-23 (VGFKL16, 52)
Function Key Lights 24-31 (VGFKL24, 53)

```
┌─────────┐
│  bits   │
└─────────┘
 0       7
```

These four registers contain a total of 32 bits, one for each of the function keys 0-31. Their status determines whether or not each key is lit. Function key 32 cannot be lit at will, but only lights when depressed.

Notice that the four registers are not contiguously numbered, thus requiring two XFERS to set all the function key lights.

15.3 CRT DISPLAY ORDERS

The VG graphical facilities are controlled by special instructions, called orders, which are sent to the VG by the META 4B. These orders can modify the contents of the VG register file, display lines and points, display characters, etc. VG orders can be generated at assembly time using macros, or at run-time. The following paragraphs describe them.

15.3.1 NO-OPERATION AND SPECIAL ORDERS

VGNOP

```
┌────┬────────────┐
│0000│////////////│
└────┴────────────┘
 0    4          15
```

VGSPEC [DATA=<value>][,PBIT=NO+ | YES]

```
┌─┬───┬──────────────┐
|P|010|    value     |
└─┴───┴──────────────┘
0 1   4             15
```

The VGNOP order perfoms no operation at the VG.

The VGSPEC order serves the special purpose of allowing the user to generate an interrupt request to the META 4B. These interrupt requests are called P-bit interrupts and are activated by the presence of bit 0 in the order. Optionally, a 12-bit integer can be included in the order, specified at assembly time by the DATA argument.

When a P-bit is encountered, the VG will stop accepting orders, and then will request an interrupt to the META 4B.

15.3.2 LOAD/ADD/AND/OR REG VALUES

VGLOAD/ADD/AND/OR <first-reg>,<value1>,...,<valuen>

```
┌─────────┬─────┬────────┐
| op      |/////|frst-rg|
└─────────┴─────┴────────┘
0         5     9       15
```

These orders can be used to modify the contents of registers in the VG register file. "n" contiguously-numbered registers are modified, starting with the one specified by <first-reg>. The registers are modified by replacing them with the values, adding the values to them, or 'AND'ing or 'OR'ing the values with them. A value can be specified as a signed fraction, a decimal number from -2048 to +2047, a 2- or 3-digit hexadecimal number, or a symbol.

These values are generated left-justified in "n" halfwords following the order. The low-order four bits of these halfwords are zero, except for the final halfword, in which bit 15 must be on. This should be remembered if the last value is to be changed dynamically.

Note that it is now possible to change VG registers with both the XFER instruction and a display order.

15.3.3 ABSOLUTE AND RELATIVE VECTOR ORDERS

```
VGABS/REL  [MODE=LINE+ | DASHED | DOTTED | POINT]
┌────┬──────┬──┬───┐
|0001|//////|md| op |
└────┴──────┴──┴───┘
 0    4      10 12 15
```

```
VGC    <value>,<op>,<axis>
┌───────────┬──┬──┐
|   value    |op|ax|
└───────────┴──┴──┘
 0           12 14
```

These orders allow the user to draw vectors on the VG scope. Vector mode is entered by specifying a VGABS or VGREL order, which is followed by an arbitrary number of VGC coordinate specifications.

VGABS mode causes absolute vectors to be drawn, that is, the user specifies absolute scope space coordinates. VGREL mode causes relative vectors to be drawn, that is, the user specifies coordinates which are relative to the current beam position just before the vector mode was entered.

Vectors can be drawn as solid (md=00), dashed (md=01), or dotted (md=10) lines, or as simple end-points (a dot at the end of the vector) (md=11).

To specify the actual vectors, the user can use the VGC macro or generate them at run-time. In either case, he must specify actual 12-bit coordinate values, whether they pertain to the X, Y, or Z axis, and how they are to be used. Coordinate values can be specified as signed fractions, integers from -2048 to +2047, 3-digit hexadecimal numbers, or symbols. The axis is specified as "X", "Y", or "Z". Coordinate operations are as follows:

"L": The coordinate value is loaded (VGABS) or added (VGREL) to the specified coordinate register, but the beam position is not changed.

"LM": The coordinate register is modified as with "L", and the beam is them moved to the position specified by all coordinate registers.

"LD": The coordinate register is modified as with "L", and the beam is then drawn to the position specified by all coordinate registers.

"LDT": Operation proceeds as with "LD". In addition, after drawing, the vector mode is terminated, and the next halfword is assumed to be a new order. "LDT" must be the last operation in a list of coordinate specifications.

When building VGC halfwords at run-time, the following method is used: compute the coordinate value and shift it into the high-order 12 bits of a work register, zeroing the low-order four bits. Then 'OR' in the <op> and <axis> bits. Mnemonic names are provided by the M4BEQUS macro for this purpose (if the "VG" argument is specified). They are "VGL", "VGLM", "VGLD", and "VGLDT" for the operations, and "VGX", "VGY", and "VGZ" for the axis. If these names are 'OR'ed into a value (with bits 12-15 off), they will set the appropriate bits.

An example may clarify the use of VGC specifications. Suppose I wanted to draw a 2-dimensional dashed line from the point (0,0) to the point (.4,.5). The following sequence could be used:

```
VGABS     MODE=DASHED
VGC       0,L,X
VGC       0,LM,Y
VGC       .4,L,X
VGC       .5,LDT,Y
```

15.3.4 CHARACTER ORDER

```
                    11          10          01          00
VGCHAR    [SIZE=16X32+ | 30X60 | 40X80 |  60X120 | PREVIOUS]
          [,SLANT=HCRIZONTAL+ | VERTICAL]
```

```
┌────┬────┬─┬───┬────┐
|0001|////|s|siz|1111|
└────┴────┴─┴───┴────┘
 0    4    8 9   12 15
```

This order causes character mode to be entered. The character size can be specified explicitly, or the size from the previous VGCHAR order can be used. If an explicit size is specified, the "siz" field contains a 1 followed by the size code. If PREVIOUS is used, it contains zeroes. The characters can be written horizontally (s=0) or vertically (s=1) (e.g., for graphs).

Following the order is an arbitrarily long string of ASCII character codes, one per byte, specifying the message to be displayed. This string can be of an odd or even length, but must end with the VGTERM control character. A macro is provided to generate ASCII character codes:

ASCII <string1>,...,<stringn>

Each string is either a string of characters enclosed in pops (pops or ampersands within this string must be doubled), or a single character code. Single characters can be specified using the symbolic names generated by the M4BEQUS macro, or as any expression.

The characters are displayed with the center of the first one lying at the current beam position. The beam position is updated as each character is displayed.

Examples:

```
     VGCHAR          SIZE=40X80
     ASCII           'DOG ','BONE '
     ASCII           'EAT',VGTERM
     VGCHAR          SIZE=PREVIOUS
     ASCII           '1.',VGCR
     ASCII           '2.',VGTERM
```

15.4 VG INTERRUPTS

15.4.1 VG UNIT CONTROL BLOCK

The VG is at unit address 9, and its UCB pointer is at location X'42', since the UCB pointer table begins at location X'30'. The q-interpreter expects the first two halfwords of the UCB to look as follows (the usual format):

VG UCB

```
┌─────────────────┐
│                9│
├─────────────────┤
│ interrupt USH   │
└─────────────────┘
0               15
```

+f grammar

The contents of the USH is irrelevant.

15.4.2 INTERRUPT SENSING

Whenever an event occurs in the VG which causes an interrupt request to the META 4B, this fact is recorded in the B's VG Interrupt Register (VGIR), control register 24. This will in turn cause an interrupt call if enabled in the ICMR (see next section).

The format of the VGIR is as follows:

```
VGIR
┌─┬─┬─┬─┬─┬─┬──────────────┐
│P│C│L│D│K│F│              │
└─┴─┴─┴─┴─┴─┴──────────────┘
 0 1 2 3 4 5             15
```

P: if on, a P-bit interrupt request has been made.

C: if on, the refresh rate clock has timed out a 40th of a second, signifying that another image frame can be drawn.

L: if on, a light pen interrupt request has been made.

D: if on, a data tablet interrupt request has been made.

K: if on, a keyboard interrupt request has been made.

F: if on, a function key interrupt request has been made.

Once an interrupt call has occurred for an interrupt request, the corresponding VGIR bit is turned off by the q-interpreter.

15.4.3 INTERRUPT CALL CONTRCL

Various types of gadgets which can cause interrupt requests have been described above. At the VG end, these requests can be controlled by the VGMCR register. At the META 4B end, the ICMR can be used to control the occurence of interrupt calls for these requests.

The high-order byte of the ICMR contains the following bits:

```
ICMR
┌─┬─┬─┬─┬─┬─┬─┬─┬──────────────┐
│P│/│L│D│K│F│/│T│              │
└─┴─┴─┴─┴─┴─┴─┴─┴──────────────┘
 0 1 2 3 4 5 6 7             15
```

P: P-bit mask. If on, P-bit interrupt requests resulting from VGSPEC orders cause interrupt calls. If off, they do not, and the request remains pending in the VGIR.

L: Light pen mask. If on, the occurrence of light pen interrupt calls is controlled by the T bit (bit 7). If the T bit is off, interrupt calls always occur. If the T bit is on, the calls occur only if the light pen tip switch is being touched (otherwise the interrupt request is discarded). NOTE that a light pen interrupt call will occur each time the pen sees light for as long as the switch is touched.

On the other hand, if the light pen mask is off, the light pen request remains pending. Note that once the interrupt call does occur, there will be no way to correlate the light pen position.

D: Data Tablet mask. If on, data tablet interrupt requests cause interrupt calls. If off, they do not, and the request remains pending.

K: Keyboard mask. If on, keyboard interrupt requests cause interrupt calls. If off, they do not, and the request remains pending.

F: Function key mask. If on, function key interrupt requests cause interrupt calls. If off, they do not, and the request remains pending.

When a VG interrupt call occurs, the interrupt procedure receives the following information:

GPR6: X'8009'

GPR7: The ICMR bit number of the type of interrupt (0 for P-bit, 2 for light pen, 3 for data tablet, 4 for keyboard, or 5 for function keys).

GPR8: Special information depending upon the type of interrupt. For keyboard interrupts, it contains the ASCII character code in bits 8-15, zeroes in bits 0-7. For function key interrupts, it contains the function key number (0-32). For the other interrupts, it contains cruft.

?

16 ET CETERA INSTRUCTION

16.1 INTRODUCTION

16.1.1 RATIONALE

In the preceeding chapters we have described the SIMALE and the Vector General. In each case, a (perhaps complex) data structure is needed to contain the data information to be interpreted by these units. In the case of the SIMALE the data might be coordinates, characters, or floating-point numbers. For the Vector General, the data is a linear sequence of orders specifying a display image. The question which remains is: how do we provide this data to these units?

The ET CETERA (ETC) instruction gives us this capability. When an ETC is executed, the META 4B q-interpreter goes into a special mode in which it acts as an interface between the user-specified data structure and the SIMALE and/or Vector General. It directs the interpretation of the data structure at the highest level, extracting information and trasferring it to and from these units.

16.1.2 ETC REGISTERS

In order for the user and the q-interpreter to control the interpretation of the ETC data structure, a set of sixteen registers is used. These are the ETC registers 32-47, also referenceable as local store locations 32-47.

Some of these registers are set up by the user before issuing an ETC instruction -- these registers specify the whereabouts of the data structure to be interpreted. The q-interpreter updates these registers as it extracts information from the structure.

Each ETC register will be described below. They are given symbolic names which are defined by the M4BEQUS macro, as usual.

16.1.3 ETC INSTRUCTION

```
ET Cetera
ETC         name
┌─────────┬─────────┬──────────────────┐
|   6F    |/////////|       name       |
└─────────┴─────────┴──────────────────┘
0         8         16                31
```

The ETC instruction causes interpretation of the user data structure to begin, as specified by certain ETC registers. The halfword name is loaded into ETC register 47, the ETCNAME register. This gives the user a means of identifying which ETC instruction was executing if one should be aborted for any reason (such as by a procedure check).

16.2 ETC DATA STRUCTURE

16.2.1 DATA AREA

The data structure for an ETC instruction lies within an area of main store called a data area. Although this data area resides in a specific place in main store, all pointers to or addresses of items within this area are relative to the start of the area. Thus the first byte in the area is relatively addressed as byte 0, while the 100th halfword would be addressed as byte 198.

The purpose of relative addressing is to provide the user an easy and efficient means of relocating his data structure without changing the actual data. An example is when she writes a data area out to disk and reads it back into a different place in main store.

When a data area is to be used initially by an ETC instruction, its address (absolute) must be placed in the Data Base Register (DBR), ETC register 32.

A data area can be created at assembly time by coding the following macros:

```
[<labela>]  DATA
    •
    • (info within data area)
    •
[<labelb>]  ENDDATA
```

If the user wishes to use absolute pointers within the data area (i.e., relative to absolute location 0), the label on the DATA macro should be ommitted. This is useful if the data is to be built at run time, but not saved.

16.2.2 BLOCKS

Within the data area, information to be interpreted is arranged in a linked ring of blocks. Each block is composed of three parts:

1. The block header. The header contains two halfwords, the first of which contains a relative pointer to the next block in the ring. The second halfword can be used for any purposes desired by the programmer (a typical use might be for a relative pointer to the previous block, making the data structure a doubly-linked ring).

2. A set of sub-blocks. It is the sub-blocks which contain the actual data to be interpreted. An arbitrary number of sub-blocks can reside in each block.

3. A halfword containing zero. This marks the end of the block.

Before an ETC instruction can be issued, the Block Pointer (BP) (ETC register 33) must be initialized with the relative address of the initial block to be interpreted within the data area.

A block can be created at assembly time by coding the following macros:

```
<labela>     BLK       next-blk[,PREV=<blk-label> |
                                        SECOND=<value>]
    •
    •  (info within block)
    •
[<labelb>]    ENDBLK
```

<next-blk> is the label on the BLK macro for the next block in the ring. If the second header halfword is to be used, it can be specified as a relative pointer (PREV=) or as an absolute value (SECOND=).

16.2.3 SUB-BLOCKS

Sub-blocks are the entities containing the actual data to be interpreted. A sub-block resides within a block, and consists of two parts:

1. The sub-block header. This header is two halfwords long, and the first halfword contains the length (in bytes) of the sub-block, including the four bytes of header. A sub-block may be odd in length, in which case the extra byte after the end is ignored. The second halfword contains information specifying how this sub-block is to be interpreted. If bit 0 is off, then this sub-block is to be interpreted by the SIMALE, and the remaining bits contain SIMALE initialization information. If bit 0 is on, then this sub-block contains only orders to be sent directly to the Vector General.

2. Sub-block data. Following the header is the data to be interpreted by the SIMALE or the orders to be sent to the Vector General.

Before an ETC is issued, the Sub-Block Pointer (SBP) (ETC register 34) must be set to contain the relative address of the initial sub-block (within the initial block) to be interpreted. This is usually the first sub-block in the initial block.

A sub-block can be created at assembly time by coding the following macros:

```
[<labela>]     SUBLK     <simale-init> | VG
               •
               • (data within sub-block)
               •
[<labelb>]     ENDSUBLK
```

The argument to SUBLK is either SIMALE initialization information or the tag "VG". The sub-block length is filled in automatically.

NOTE: The SUBLK macro generates labels in the form SUBn (n=1,2,3...). The user should avoid using such labels.

16.2.4 ETC INSTRUCTION EXECUTION

When an ETC instruction is issued, after setting up the DBR, BP, and SBP, execution proceeds as follows:

1. Starting with the initial block and sub-block, interpret all the sub-blocks in each block, switching

to new blocks each time the ending zero halfword is encountered.

2. If, while interpreting, an error condition should arise (e.g., an odd next-block offset), abort execution and cause the appropriate procedure check.

3. Otherwise, if a META 4A, SIMALE, or Vector General interrupt call is required (e.g., due to an enabled light pen hit), abort execution so that it may occur.

4. Otherwise, if none of the above occur, display the entire data ring exactly once, and then abort.

Whenever we abort ETC execution, the BP and SBP will be updated to point at the next block/sub-block that would have been displayed had we not aborted. This means the programmer has to set up these registers only once, and then can issue multiple ETCs (perhaps in a loop). Each successive ETC will take up wherever the previous one left off.

Furthermore, four other ETC registers will be set to indicate where we were interpreting when the abort occurred. These are the Final Data Base Register (FDBR), Final Block Pointer (FBP), Final Sub-Block Pointer (FSBP), and Final Data Pointer(FDP), ETC registers 40-43. These can tell the programmer in which data area, block, and sub-block the abort occurred, plus which halfword was the last one interpreted. The FDBR contains an absolute address, while the others are relative to it.

16.3 VECTOR GENERAL SUB-BLOCKS

During interpretation of Vector General sub-blocks, the ETC instruction may be aborted due to errors or the completion of one circuit of the block ring. In addition, if a P-bit or light pen interrupt request is recognized, interpretation will be immediately aborted so that an interrupt call can occur. Keyboard or function key interrupts will not abort -- they are left pending during ETC execution. *

After an abort due to a P-bit request, the FDP will point at the VGSPEC order in question. After one due to a light pen hit, the FDP will be accurate if a vector was being displayed, but may be off by one or two halfwords if characters were.

16.4 THE CLOCK INSTRUCTION

Clearly, one of the principal uses of the ETC instruction is to perform graphical data manipulation and display. When using the Vector General scope, it is necessary to ensure that simple images are not displayed too often. For example, if an ETC instruction were to be used to display a single dot, and this ETC were to be put in a tight loop in order to maintain the image, a wonderful burned spot would soon appear in the phosphorus.

As explained in Chapter 15.1.2, the Vector General has a refresh rate clock which pulses 40 times per second. No image should be displayed more often that this. Hence the programmer needs an instruction which pauses until the next clock pulse:

```
CLOCK vector general
CLOCK     0
┌────────┬──────────────────────────────┐
|   8E   |//////////////////////////////|
└────────┴──────────────────────────────┘
0        8                             31
```

A CLOCK instruction should appear at the top of every display loop using ETC.

16.5 A TYPICAL DISPLAY SEQUENCE

The following is an outline of a typical sequence employing the ETC instruction to display an image upon the Vector General:

```
        set up initial DBR, BP, and SBP
LOOP    CLOCK 0
        update dynamic portion of data structure
        ETC     0
        test for terminate conditions
        GOTO LOOP if not satisfied
```

17 RUNNING META 4B PROGRAMS -- MULTIPAC

When running programs on BUGS using the META 4B, a monitor is needed to control such things as META 4A/B communication, META 4B interrupts, main store allocation by the META 4B, etc. This monitor is called MULTIPAC, and is controlled by the user via the MULTI macro.

Two modes of programming are available with MULTIPAC:

1. RUNB mode. In RUNB mode, it is assumed that the user wants to write code only for the META 4B; no A programs will be present. A "null" A program is provided which will load and execute any such B program.

2. Normal mode. In this mode, the user writes both A and B programs, and they communicate via q-interpreter and MULTIPAC facilities.

17.1 RUNB MODE

In RUNB mode, the user writes code only for the META 4B. The mainline B program must use the MULTI macro to set up an operating environment and to specify Vector General interrupt handling options, if required. This macro should be the first thing executed in the mainline and is coded as follows:

```
MULTI META4B[,VGINT=<procn>
            [,ICMR=(P-BIT+,
                    LIGHT-PEN+,
                    DATA-TABLET+,
                    KEYBOARD+,
                    FUNCTION-KEYS+,
                    TIP-SWITCH+,
                    META-4A+)
            [,VGMCR=(LIGHT-PEN+,
                     DATA-TABLET+,
                     KEYBOARD+,
                     FUNCTION-KEYS+,
                     BLINK)]]]
```

If Vector General interrupts are to be accepted, the VGINT argument specifies a procedure to handle them. The ICMR argument specifies which and how interrupt calls are to be enabled in the B's ICMR. The VGMCR argument specifies the type of interrupts or opticn to be enabled in the VG MCR register. Any combinations of keywords, in any order, is allowed.

Upon return from the MULTI macro, interrupts will be enabled as specified by the ICMR and VGMCR options. Enabling may be changed later if the user so desires.

When the mainline B program determines that execution is complete, the following macro should be executed to terminate processing:

```
     RUNBDONE
```

This generates code to inform MULTIPAC of completion and to return from the B mainline.

When you are ready to run a program in RUNB mode, generate a BUGS MODU file under CMS using the GMSLINK command.

Once the MODU is on the BUGS disk, it can be run with the following GMS command:

```
RUNB <modu-name>
```

* A sample RUNB mode program is shown in Appendix 4a.

17.2 NORMAL MODE

In normal mode, the user writes programs for both machines, which can communicate using the facilities described in Chapter 13. When a normal mode program is executed, the A mainline gets control first. It must then start up a B mainline, which will run in parallel with it. To do this, another version of the MULTI macro is used:

Is this
this A
or B
pgm??
(address A)

```
     MULTI     START,M4BPROC=<procn>
                    [,M4BINT=<procn>
                    [,MAXUSH=<expression>]]
```

The M4BPROC argument specifies the address of the B mainline. If B interrupts are to be accepted by the A (these would be * generated by an INTA instruction), M4BINT specifies the address of an A procedure to handle them. The MAXUSH argument can be used to set a limit on the interrupt code, so that all codes above this limit are discarded. The default limit is X'3FFF'.

The M4BINT procedure, if present, must begin with an ENT and return with a RET. The interrupt code from the B will be in register 2 upon entry.

Additional arguments are available on the MULTI META4B macro to specify handling of A interrupts. These are:

```
MULTI     META4B,...[,M4AINT=<procn>
                     [,MAXUSH=<expression>]]
```

If A interrupts are to be accepted by the B (these would be generated by an INTB instruction), the M4AINT argument specifies a procedure to handle them. The MAXUSH argument corresponds to the one on the MULTI START macro.

The M4AINT procedure, if present, must be a standard B procedure. The interrupt code from the A will be in GPR7 upon entry.

When the mainline A program determines that execution is complete, it should issue the following macro __before__ POSTing GMS and RETurning:

```
MULTI     DONE
```

On the other hand, when the mainline B program desires to complete, all it needs to do is RETURN. The two mainline __must__ complete; they may complete in any order -- MULTIPAC waits until they are both finished.

A and B programs may of course be GMSLINKed into the same MODU files, and they may refer to each other via V-constants.

A sample normal mode program is shown in Appendix 4b.

__17.3 MAIN STORE CONTROL IN THE B__

As in the A, a user of the B can allocate and free blocks of main store. An allocated block is always aligned on a halfword boundary, and is always a multiple of four bytes in length. To allocate a block, the B user codes:

```
MULTI     ALLOCATE,SIZE=<size>,ADDRESS=<gpr>
```

The <size> can be specified as an XBD address or as a GPR in parenthases. This size is rounded __up__ to a multiple of four, space is obtained from main store, and its address is returned in the GPR specified by the ADDRESS argument. If space is not available, an address of zero is returned.

When the user no longer needs this allocated space, __it should be freed__. To do this, the user codes:

```
MULTI     FREE,ADDRESS=<gpr>,SIZE=<size>
```

The size is again rounded, and the secified block is deallocated.

17.4 GMS SVCS FROM THE B

MULTIPAC provides a facility which allows the user to execute GMS SVCs from the B. Thus the user could execute commands, manipulate files, or read cards from the B. This is especially useful in RUNB mode. To do this, one codes:

```
     MULTI        GMSVC,SVC=<expression>,ARGLIST=<label>,
                            WAIT=YES+ | NO
```

The SVC specified by the SVC argument is executed, using the argument list specified by ARGLIST. This argument list should be identical to the one used on the A.

In order to wait for completion of the SVC, the user can code WAIT=YES, or can test for completion later in the program (WAIT=NO). To test for completion, code:

```
     WAIT     <arglist-label>
```

This loops until the WCH is POSTed by the A.

18 THE FUDD DEBUGGING PACKAGE

The FUDD debugging package is described in a separate document entitled "FUDD: Interactive Debugger Users' Guide".

APPENDIX 0: METALINGUISTIC SYMBOLS

- Syntactic constants are specified in upper-case letters.

- Syntactic variables are enclosed in angle brackets ("<" and ">"), and named using lower-case letters.

- Optional items are enclosed in square brackets ("[" and "

- The minimum abbreviation for a keyword is underscored.

- Defaults are specified by following them with a superscript plus sign ("+").

- Syntactic variables followed by a box ("□") are special address specifications. They can be coded as a label, a V-constant, or a register in parenthases.

APPENDIX 2A: INSTRUCTIONS BY MNEMONIC

MNEMONIC	OPCODE		SETS PAGECFR?	
ABSH	9F6		NO	41
ABSR	0F6		NO	41
AHA	C1		YES	43
AHH	D1		YES	43
AHI	B1		YES	43
AHR	91		YES	42
ARA	71		YES	42
ARH	61		YES	42
ARI	51		YES	42
ARR	01		YES	42
B	4		NO	22
BC	5D8	1	NO	43
BCF	5D		NO	24
BE	5D8	2	NO	51
BF	2		NO	24
BG	5D4	3	NO	51
BH	9FC		NO	22
BL	5D2	4	NO	51
BM	5D2	5	NO	58
BN	5D2	6	NO	38
BNE	5D6	7	NO	51
BNG	5DA	8	NO	51
BNL	5DC	9	NO	51
BNM	5DC	10	NO	58
BNN	5DC	11	NO	38
BNO	5DA	12	NO	58
BNP	5DA	13	NO	38
BNS	5D4	14	NO	52
BNZ	5D6	15	NO	38,58
BO	5D4	16	NO	43,58
BP	5D4	17	NO	38
BR	0FC		NO	22
BS	5D8	18	NO	58
BT	3		NO	24
BZ	5D8	19	NO	38,58
CALLA	7FD		NO	64
CALLH	9FD		NO	64
CALLR	0FD		NO	64
CASE	6D		NO	25
CHA	C5		YES	51
CHH	D5		YES	51
CHI	B5		YES	51
CHR	95		YES	50
CLBI	EB		YES	53
CLBR	AB		YES	53
CLCC	FB		YES	53
CLHA	CB		YES	53
CLHH	DB		YES	53
CLHI	BB		YES	53

CLHR	9B	YES	52
CLOCK	8E	NO	95
CLRA	7B	YES	52
CLRB	8B	YES	52
CLRH	6B	YES	52
CLRI	5B	YES	52
CLRR	0B	YES	52
CRA	75	YES	50
CRH	65	YES	50
CRI	55	YES	50
CRR	05	YES	50
DDTSH	9F5	YES	40
DDTSR	0F5	YES	40
DRA	74	YES	47
DRH	64	YES	47
DRI	54	YES	47
DRR	04	YES	47
DTSH	9F4	YES	40
DTSR	0F4	YES	40
ETC	6F	MUNG	91
EXSR	0F9	NO	48
IDLE	8D	NO	***
IH	9F2	NO	39
IIH	9F3	NO	39
IIR	0F3	NO	39
INIT	FE	NO	60
INTAA	7FF	NO	73
INTAH	9FF	NO	73
INTAR	0FF	NO	73
IR	0F2	NO	39
LSHDI	16	YES	34
LSHDR	1E	YES	34
LSHI	14	YES	33
LSHLDI	12	NO	37
LSHLDR	1A	NO	37
LSHLI	10	NO	36
LSHLR	18	NO	36
LSHR	1C	YES	33
MRA	73	NO	45
MRH	63	NO	45
MRI	53	NO	45
MRR	03	NO	45
NBI	E9	NO	57
NBR	A9	NO	57
NCC	F9	NO	57
NEGH	9F7	NO	41
NEGR	0F7	NO	41
NHA	C9	NO	57
NHH	D9	NO	57

NHI	B9	NO	57
NHR	99	NO	57
NOP	0D0	NO	22
NRA	79	NO	57
NRB	89	NO	57
NRH	69	NO	57
NRI	59	NO	57
NRR	09	NO	57
OBI	E8	NO	56
OBR	A8	NO	56
OCC	F8	NO	56
OHA	C8	NO	56
OHH	D8	NO	56
OHI	B8	NO	56
OHR	98	NO	56
ORA	78	NO	55
ORB	88	NO	55
ORH	68	NO	55
ORI	58	NO	55
ORR	08	NO	55
RBI	E6	NO	29
RBR	A6	NO	29
RCC	F6	NO	29
RETURN	5E	NO	67
RHA	C6	NO	29
RHH	D6	NO	29
RHI	B6	NO	29
RHR	96	NO	28
RRA	76	NO	28
RRB	86	NO	28
RRH	66	NO	28
RRI	56	NO	28
RRR	06	NO	28
RSHDI	17	NO	35
RSHDR	1F	NO	35
RSHI	15	NO	33
RSHLDI	13	NO	37
RSHLDR	1B	NO	37
RSHLI	11	NO	36
RSHLR	19	NO	36
RSHR	1D	NO	34
SCBEA	F2	YES	58
SCBNA	F3	YES	59
SCBT	F5	YES	53
SCFEA	F0	YES	58
SCFNA	F1	YES	59
SCFT	F4	YES	59
SEMA	9D0	YES	74
SHA	C2	YES	44
SHH	D2	YES	44
SHI	B2	YES	44
SHR	92	YES	44

SIMALE	5F	NO	***
SINTA	7FE	NO	69
SINTH	9FE	NO	69
SINTR	0FE	NO	69
SOH	9F1	NO	26
SOR	0F1	NO	26
SQRTH	9FA	NO	49
SQRTR	0FA	NO	48
SRA	72	YES	44
SRH	62	YES	44
SRI	52	YES	44
SRR	02	YES	44
SWBR	A7	NO	31
SWHH	D7	NO	31
SWHR	97	NO	31
SWRB	87	NO	31
SWRH	67	NO	31
SWRR	07	NO	31
SZH	9F0	NO	26
SZR	0F0	NO	26
TMBI	EC	YES	57
TMBR	AC	YES	57
TMHA	CC	YES	57
TMHH	DC	YES	57
TMHI	BC	YES	57
TMHR	9C	YES	57
TMRA	7C	YES	57
TMRB	8C	YES	57
TMRH	6C	YES	57
TMRI	5C	YES	57
TMRR	0C	YES	57
TR	FD	NO	60
TSA	7F8	YES	38
TSH	9F8	YES	38
TSR	0F8	YES	38
XBI	EA	NO	57
XBR	AA	NO	57
XCC	FA	NO	57
XFER	FF	NO	19
XHA	CA	NO	57
XHH	DA	NO	57
XHI	BA	NO	57
XHR	9A	NO	57
XRA	7A	NO	57
XRB	8A	NO	57
XRH	6A	NO	57
XRI	5A	NO	57
XRR	0A	NO	57

APPENDIX 2B: INSTRUCTIONS BY OPERATION CODE

CP CODE	MNEMONIC	SETS CFR?
01	ARR	YES
02	SRR	YES
03	MRR	NO
04	DRR	YES
05	CRR	YES
06	RRR	NO
07	SWRR	NO
08	ORR	NO
09	NRR	NO
0A	XRR	NO
0B	CLRR	YES
0C	TMRR	YES
0D0	NOP	NO
0F0	SZR	NO
0F1	SOR	NO
0F2	IR	NO
0F3	IIR	NO
0F4	DTSR	YES
0F5	DDTSR	YES
0F6	ABSR	NO
0F7	NEGR	NO
0F8	TSR	YES
0F9	EXSR	NO
0FA	SQRTR	NO
0FC	BR	NO
0FD	CALLR	NO
0FE	SINTR	NO
0FF	INTAR	NO
10	LSHLI	NO
11	RSHLI	NO
12	LSHLDI	NO
13	RSHLDI	NO
14	LSHI	YES
15	RSHI	NO
16	LSHDI	YES
17	RSHDI	NO
18	LSHLR	NO
19	RSHLR	NO
1A	LSHLDR	NO
1B	LDR	NO
1C	LSHR	YES
1D	RSHR	NO
1E	LSHDR	YES
1F	RSHDR	NO
20	BF	NO
30	BT	NO

0D
0DX
0E
0F8
50
57
5DE
60
6E
70
77
7D
7E
7F k ≠ 8, D, B, F
80
81
82
83
84
85
8F
90
93
94
9D ≠ 0
9F9
9FB
A0
A1
A2
A3
A4
A5
A9
AC
AF
B0 C0 D0
B3 C3 D3
B4 C4 D4
B7 C7
BD CD DD
BE CE DE
BF CF DF
E0
E1
E2
E3
E4
E5
E7 F7
ED FC
EE
EF

Appendix 2b - Instructions by Opcodes

40	B	NO
51	ARI	YES
52	SRI	YES
53	MRI	NO
54	DRI	YES
55	CRI	YES
56	RRI	NO
58	ORI	NO
59	NRI	NO
5A	XRI	NO
5B	CLRI	YES
5C	TMRI	YES
5D	BCF	NO
5D2	BM	NO
5D2	BN	NO
5D2	BG	NO
5D4	BP	NO
5D4	BO	NO
5D4	BL	NO
5D4	BNS	NO
5D6	BNZ	NO
5D6	BNE	NO
5D8	BZ	NO
5D8	BC	NO
5D8	BE	NO
5D8	BS	NO
5DA	BNP	NO
5DC	BNL	NO
5DA	BNO	NO
5DC	BNM	NO
5DC	BNN	NO
5DC	BNG	NO
5E	RETURN	NO
5F	SIMALE	NO
61	ARH	YES
62	SRH	YES
63	MRH	NO
64	DRH	YES
65	CRH	YES
66	RRH	NO
67	SWRH	NO
68	ORH	NO
69	NRH	NO
6A	XRH	NO
6B	CLRH	YES
6C	TMRH	YES
6D	CASE	NO
6F	ETC	MUNG
71	ARA	YES
72	SRA	YES
73	MRA	NO
74	DRA	YES

75	CRA	YES	
76	RRA	NO	
78	ORA	NO	
79	NRA	NO	
7A	XRA	NO	
7B	CLRA	YES	
7C	TMRA	YES	
7F8	TSA	YES	
7FD	CALLA	NO	
7FE	SINTA	NO	
7FF	INTAA	NO	
86	RRB	NO	
87	SWRB	NO	
88	ORB	NO	
89	NRB	NO	
8A	XRB	NO	
8B	CLRB	YES	
8C	TMRB	YES	
8D	IDLE	NO	
8E	CLOCK	NO	
91	AHR	YES	
92	SHR	YES	
95	CHR	YES	
96	RHR	NO	
97	SWHR	NO	
98	OHR	NO	
99	NHR	NO	
9A	XHR	NO	
9B	CLHR	YES	
9C	TMHR	YES	
9D0	SEMA	NO	What?!
9F0	SZH	NO	
9F1	SOH	NO	
9F2	IH	NO	
9F3	IIH	NO	
9F4	DTSH	YES	
9F5	DDTSH	YES	
9F6	ABSH	NO	
9F7	NEGH	NO	
9F8	TSH	YES	
9FA	SQRTH	NO	
9FC	BH	NO	
9FD	CALLH	NO	
9FE	SINTH	NO	
9FF	INTAH	NO	
A6	RBR	NO	
A7	SWBR	NO	
A8	OBR	NO	
A9	NBR	NO	
AA	XBR	NO	
AB	CLBR	YES	
AC	TMBR	YES	

B1	AHI	YES
B2	SHI	YES
B5	CHI	YES
B6	RHI	NO
B8	OHI	NO
B9	NHI	NO
BA	XHI	NO
BB	CLHI	YES
BC	TMHI	YES
C1	AHA	YES
C2	SHA	YES
C5	CHA	YES
C6	RHA	NO
C8	OHA	NO
C9	NHA	NO
CA	XHA	NO
CB	CLHA	YES
CC	TMHA	YES
D1	AHH	YES
D2	SHH	YES
D5	CHH	YES
D6	RHH	NO
D7	SWHH	NO
D8	OHH	NO
D9	NHH	NO
DA	XHH	NO
DB	CLHH	YES
DC	TMHH	YES
E6	RBI	NO
E8	OBI	NO
E9	NBI	NO
EA	XBI	NO
EB	CLBI	YES
EC	TMBI	YES
F0	SCFEA	YES
F1	SCFNA	YES
F2	SCBEA	YES
F3	SCBNA	YES
F4	SCFT	YES
F5	SCBT	YES
F6	RCC	NO
F8	OCC	NO
F9	NCC	NO
FA	XCC	NO
FB	CLCC	YES
FD	TR	NO
FE	INIT	NO
FF	XFER	NO

APPENDIX 3A: ASCII TO EBCDIC CONVERSION

ASCII	EBCDIC	GRAPHIC	ASCII	EBCDIC	GRAPHIC
00	00	* NUL	20	40	* SPACE (VG-BLANK)
01	01	SOH	21	5A	!
02	02	STX	22	7F	"
03	03	ETX	23	7B	#
04	37	EOT	24	5B	$
05	2D	ENQ	25	6C	%
06	2E	ACK	26	50	&
07	2F	BEL	27	7D	'
08	16	* BS	28	4D	(
09	05	HT	29	5D	)
0A	25	* LF	2A	5C	*
0B	0B	VT	2B	4E	+
0C	0C	FF	2C	6B	,
0D	15	* CR	2D	60	-
0E	0E	SO	2E	4B	.
0F	0F	SI	2F	61	/
10	10	DLE	30	F0	0
11	11	DC1	31	F1	1
12	12	* DECSZ	32	F2	2
13	13	* INCSZ	33	F3	3
14	3C	* TERM	34	F4	4
15	3D	NAK	35	F5	5
16	32	SYN	36	F6	6
17	26	ETB	37	F7	7
18	18	CAN	38	F8	8
19	19	EM	39	F9	9
1A	3F	SUB	3A	7A	:
1B	27	ESC	3B	5E	;
1C	1C	FS	3C	4C	<
1D	1D	GS	3D	7E	=
1E	1E	RS	3E	6E	>
1F	1F	VS	3F	6F	?

Appendix 3a - ASCII to EBCDIC

40	7C	@	60	78	`
41	C1	A	61	81	a
42	C2	B	62	82	b
43	C3	C	63	83	c
44	C4	D	64	84	d
45	C5	E	65	85	e
46	C6	F	66	86	f
47	C7	G	67	87	g
48	C8	H	68	88	h
49	C9	I	69	89	i
4A	D1	J	6A	91	j
4B	D2	K	6B	92	k
4C	D3	L	6C	93	l
4D	D4	M	6D	94	m
4E	D5	N	6E	95	n
4F	D6	O	6F	96	o
50	D7	P	70	97	p
51	D8	Q	71	98	q
52	D9	R	72	99	r
53	E2	S	73	A2	s
54	E3	T	74	A3	t
55	E4	U	75	A4	u
56	E5	V	76	A5	v
57	E6	W	77	A6	w
58	E7	X	78	A7	x
59	E8	Y	79	A8	y
5A	E9	Z	7A	A9	z
5B	AD	[	7B	8B	{
5C	E0	\	7C	4F	| (¦)
5D	BD	]	7D	9B	}
5E	71	↑	7E	59	~
5F	6D	_	7F	07	DEL

Appendix 3a - ASCII to EBCDIC

80	20	* (CUP)	A0	43	□
81	21	* (CDOWN)	A1	44	↓
82	22	* (CHOME)	A2	48	‖
83	23	* (CFOR)	A3	75	○
84	24	* (CBACK)	A4	DB	Ƶ
85	CB		A5	51	V
86	9F		A6	41	
87	77		A7	53	√
88	E0		A8	54	⊂
89	9F		A9	55	⊃
8A	9F		AA	58	
8B	9F	▥	AB	62	÷
8C	9F		AC	8C	≤
8D	9F		AD	64	≡
8E	9F		AE	AE	≥
8F	9F		AF	5F	¬
90	9F		B0	9F	o
91	A1		B1	45	↑
92	61	/	B2	67	∀
93	9F		B3	76	□
94	9F		B4	4A	¢
95	9F		B5	52	
96	CB		B6	9F	
97	9F		B7	68	
98	9F		B8	57	∪
99	61	/	B9	56	∩
9A	9F		BA	49	∘
9B	41		BB	63	×
9C	9F		BC	47	←
9D	42	E	BD	BE	≠
9E	9F		BE	46	→
9F	9F		BF	69	∞

C0	72	¨	E0	* 15	(CURSOR)
C1	67	∀	E1	AA	α
C2	BF	_ (underscore)	E2	FC	β
C3	9F		E3	6A	∇
C4	8D	Δ	E4	B0	δ
C5	66	∃	E5	B1	ϵ
C6	8E	Φ̲	E6	B2	
C7	AF	∘	E7	FD	γ
C8	ED	⊤	E8	EC	⊥
C9	8F	ψ	E9	BC	
CA	80		EA	E0	
CB	9F		EB	74	
CC	9F		EC	B3	λ
CD	9F	∧	ED	B4	μ
CE	90		EE	B5	
CF	9C	Ω	EF	B6	ω
D0	BB	π	F0	B7	π
D1	9F		F1	65	
D2	9D		F2	B8	ρ
D3	9E	Σ	F3	B9	σ
D4	A0	θ	F4	BA	τ
D5	9F		F5	9F	
D6	70		F6	9F	
D7	9F		F7	9F	
D8	9F		D8	9F	
D9	9F		? F9	68	∠
DA	CB		FA	* 04	SCROTUM (VGSCROT)
DB	AB	⌊	FB	AC	⌈
DC	73	⇒	FC	4F	|
DD	EE	⊢	FD	EF	⊣
DE	D0		FE	C0	~
DF	A1	∘	FF	* 9F	▥ (VGBLOTZ)

APPENDIX 3B: EBCDIC TO ASCII CONVERSION

EBCDIC	ASCII	GRAPHIC	EBCDIC	ASCII	GRAPHIC
00	00	NULL	20	80	(CUP)
01	01	SOH	21	81	(CDOWN)
02	02	STX	22	82	(CHOME)
03	03	ETX	23	83	(CFOR)
04	~~04~~ FA	SCROTUM	24	84	(CBACK)
05	09	HT	25	0A	LF
06	FF		26	17	ETB
07	7F	DEL	27	1B	ESC
08	FF		28	FF	
09	FF		29	FF	
0A	FF		2A	FF	
0B	0B	VT	2B	FF	
0C	0C	FF	2C	FF	
0D	FF		2D	05	ENQ
0E	0E	SO	2E	06	ACK
0F	0F	SI	2F	07	BEL
10	10	DLE	30	FF	
11	11	DC1	31	FF	
12	12	DECSZ	32	16	SYN
13	13	INCSZ	33	FF	
14	E0	(CURSOR)	34	FF	
15	0D	CR	35	FF	
16	08	BS	36	FF	
17	FF		37	04	EOT
18	18	CAN	38	FF	
19	19	EM	39	FF	
1A	FF		3A	FF	
1B	FF		3B	FF	
1C	1C	FS	3C	14	TERM
1D	1D	GS	3D	15	NAK
1E	1E	RS	3E	FF	
1F	1F	US	3F	1A	SUB

Appendix 3b - EBCDIC to ASCII

EBCDIC	ASCII	GRAPHIC	EBCDIC	ASCII	GRAPHIC
40	20	SPACE	60	2D	-
41	A6		61	2F	/
42	9D		62	AB	
43	A0		63	BB	
44	A1		64	AD	
45	B1		65	F1	
46	BE		66	C5	
47	BC		67	C1	
48	A2		68	B7	
49	BA		69	BF	
4A	B4	¢	6A	E3	
4B	2E	.	6B	2C	,
4C	3C	<	6C	25	%
4D	28	(	6D	5F	_
4E	2B	+	6E	3E	>
4F	FC	|	6F	3F	?
50	26	&	70	D6	
51	A5		71	5E	
52	B5		72	C0	
53	A7		73	DC	
54	A8		74	EB	
55	A9		75	A3	
56	B9		76	B3	
57	B8		77	87	
58	AA		78	60	
59	7E		79	FF	
5A	21	!	7A	3A	:
5B	24	$	7B	23	#
5C	2A	*	7C	40	@
5D	29	)	7D	27	'
5E	3B	;	7E	3D	=
5F	AF	!	7F	22	"

Appendix 3b - EBCDIC to ASCII

EBCDIC	ASCII	GRAPHIC	EBCDIC	ASCII	GRAPHIC
80	CA		A0	D4	
81	61	a	A1	DF	
82	62	b	A2	73	s
83	63	c	A3	74	t
84	64	d	A4	75	u
85	65	e	A5	76	v
86	66	f	A6	77	w
87	67	g	A7	78	x
88	68	h	A8	79	y
89	69	i	A9	7A	z
8A	FF		AA	E1	
8B	7B		AB	DB	
8C	AC		AC	FB	
8D	C4		AD	5B	
8E	C6		AE	AE	
8F	C9		AF	C7	
90	CE		B0	E4	
91	6A	j	B1	E5	
92	6B	k	B2	E6	
93	6C	l	B3	EC	
94	6D	m	B4	ED	
95	6E	n	B5	EE	
96	6F	o	B6	EF	
97	70	p	B7	F0	
98	71	q	B8	F2	
99	72	r	B9	F3	
9A	FF		BA	F4	
9B	7D		BB	D0	
9C	CF		BC	E9	
9D	~~CF~~ D2		BD	5D	
9E	D3		BE	BD	
9F	FF		BF	C2	

EBCDIC	ASCII	GRAPHIC	EBCDIC	ASCII	GRAPHIC
C0	FE		E0	5C	
C1	41	A	E1	FF	
C2	42	B	E2	53	S
C3	43	C	E3	54	T
C4	44	D	E4	55	U
C5	45	E	E5	56	V
C6	46	F	E6	57	W
C7	47	G	E7	58	X
C8	48	H	E8	59	Y
C9	49	G	E9	5A	Z
CA	FF		EA	FF	
CB	96		EB	FF	
CC	FF		EC	E8	
CD	FF		ED	C8	
CE	FF		EE	DD	
CF	FF		EF	FD	
D0	DE		F0	30	0
D1	4A	J	F1	31	1
D2	4B	K	F2	32	2
D3	4C	L	F3	33	3
D4	4D	M	F4	34	4
D5	4E	N	F5	35	5
D6	4F	O	F6	36	6
D7	50	P	F7	37	7
D8	51	Q	F8	38	8
D9	52	R	F9	39	9
DA	FF		FA	FF	
DB	A4		FB	FF	
DC	FF		FC	E2	
DD	FF		FD	E7	
DE	FF		FE	FF	
DF	FF		FF	FF	

We would like to thank the following people for their invaluable help in the creation of the system described in this document:

Russell Wayne Burns

who designed and implemented the FUDD debugging package.

William Benjamin Rothman

who read this nonsense relentlessly, proofing and suggesting changes, some of which were inadvertantly included.

Paul Constantine Anagnostopoulos
Harold Henry Webber, Jr.
John Zahorjan

A survey of the vast experimental literature on the effects of computing system design is likely to convince the most dispassionate observer that the possibility of de-kludging is improbable.

--adapted frcm I. Steele Russell, 1971